Vrije Universiteit Brussel

Doctoral Dissertation

Static Analysis for Quality Assurance of Ansible Infrastructure-as-Code Artefacts

---

Ruben Opdebeeck

Dissertation submitted in fulfilment of the requirements for the degree of Doctor of Sciences

Promotor:
Prof. Dr. Coen De Roover

Jury:
Prof. Dr. Viviane Jonckers, Vrije Universiteit Brussel, Belgium (chair)
Prof. Dr. Antonio Paolillo, Vrije Universiteit Brussel, Belgium (secretary)
Prof. Dr. João F. Ferreira, Universidade de Lisboa, Portugal
Prof. Dr. Dimitris Mitropoulos, National and Kapodistrian University of Athens, Greece
Prof. Dr. Beat Signer, Vrije Universiteit Brussel, Belgium
Prof. Dr. Kris Steenhaut, Vrije Universiteit Brussel, Belgium

Faculty of Sciences and Bioengineering Sciences
Department of Computer Science
Software Languages Laboratory

October 2024

Printed by
Crazy Copy Center Productions
VUB Pleinlaan 2, 1050 Brussel
Tel: +32 2 629 33 44
crazycopy@vub.be
www.crazycopy.be

ISBN: 9789464948608
NUR: 980
THEMA: UMZ

The work in this dissertation has been funded by a PhD fellowship of the Research Foundation Flanders (FWO) (grant number 1SD4321N/1SD4323N) and travel grant of the Research Foundation Flanders (FWO) (grant number V431423N).

Abstract

Today's software-intensive industry uses increasingly complex digital computing infrastructures, possibly comprising hundreds of cloud resources such as virtual computing instances and managed databases. Manually managing such infrastructures is laborious and error-prone. Therefore, Infrastructure as Code (IaC) enables practitioners to automate the provisioning, configuration, and orchestration of infrastructures through executable code. The correctness and security of such code are vital, as defects can cause disastrous outages while security weaknesses leave the infrastructure vulnerable to cyberattacks. However, existing code quality assurance analyses for IaC are lacklustre and either underperform by not reasoning about the code's behaviour or are resource-intensive to apply. Moreover, they focus solely on the infrastructure code and ignore its supporting software supply chains, which form a major cyberattack vector today.

This thesis aims to alleviate these issues through four contributions aimed at Ansible, one of the most widely-used IaC tools. The first is the Program Dependence Graph (PDG) for Ansible, a graph representation of infrastructure code that captures its behaviour. We introduce an analysis to extract a PDG for Ansible scripts statically, i.e., without executing the code, thereby bridging the gap between lightweight yet underperforming approaches and accurate yet resource-intensive approaches.

The second contribution is an application of the PDG representation to detect behavioural "code smells" that may be indicative of defects in infrastructure code. By detecting patterns in these graphs, the analysis can uncover six error-prone coding practices related to Ansible variables. We apply this analysis in a large-scale empirical study to investigate the prevalence and lifetime of these smells in open-source Ansible code.

GASEL, the third contribution, is an application of the PDG representation to detect security weaknesses in IaC. It uses graph queries on the PDG to identify seven types of "security smells" that may lead to security weaknesses or attack vectors. The behavioural information encoded in the graphs enables GASEL to outperform state-of-the-art security smell detectors. We apply GASEL in a large-scale empirical study investigating security smells in open-source Ansible code.

The fourth contribution is an empirical study of Ansible's deployment software supply chain. We propose an automated analysis that identifies third-party dependencies of Ansible plugins, and we apply it to study the types of dependencies that occur in practice.

The empirical studies presented in this thesis call for improvements to quality assurance practices in Infrastructure as Code, while the proposed static analyses pave the way for advanced tooling to this end.

Samenvatting

De huidige software-intensieve industrie gebruikt steeds complexere digitale computerinfrastructuren, mogelijk bestaande uit honderden clouddiensten zoals virtuele computerinstanties en beheerde databases. Het handmatig beheren van dergelijke infrastructuren is arbeidsintensief en foutgevoelig. Daarom stelt Infrastructure as Code (IaC) beoefenaars in staat om de voorziening, configuratie en orkestratie van infrastructuren te automatiseren met uitvoerbare code. De correctheid en veiligheid van dergelijke code is cruciaal, omdat defecten rampzalige onderbrekingen kunnen veroorzaken, terwijl beveiligingszwaktes de infrastructuur kwetsbaar maken voor cyberaanvallen. Bestaande kwaliteitsborgingsanalyses voor IaC zijn echter gebrekkig en presteren ofwel ondermaats door niet te redeneren over het gedrag van de code, of zijn zeer middelenintensief om toe te passen. Bovendien richten ze zich alleen op de infrastructuurcode en negeren ze de ondersteunende softwaretoevoerketens, die tegenwoordig een belangrijke vector voor cyberaanvallen vormen.

Deze thesis tracht deze problemen aan te kaarten met vier bijdragen gericht op Ansible, een van de meest gebruikte IaC tools. De eerste is de Program Dependence Graph (PDG) voor Ansible, een graafvoorstelling van infrastructuurcode die diens gedrag omvat. We introduceren een analyse om een PDG voor Ansible-scripts statisch te extraheren, d.w.z. zonder de code uit te voeren, en overbruggen hierdoor de kloof tussen eenvoudige maar ondermaatse aanpakken en nauwkeurige maar middelenintensieve aanpakken.

De tweede bijdrage is een toepassing van de PDG-voorstelling om semantische "codegeuren" te detecteren die kunnen duiden op defecten in infrastructuurcode. Door patronen in de grafen te detecteren, kan de analyse zes foutgevoelige codeerpraktijken gerelateerd aan Ansible-variabelen blootleggen. We passen deze analyse toe in een grootschalig empirisch onderzoek om de prevalentie en levensduur van deze geuren in open-source Ansible-code te onderzoeken.

GASEL, de derde bijdrage, is een toepassing van de PDG-voorstelling om beveiligingszwaktes in IaC te detecteren. Het gebruikt graafqueries op de PDG om zeven soorten "beveiligingsgeuren" te identificeren die kunnen leiden tot beveiligingszwaktes of aanvalsvectoren. De gedragsinformatie omvat in de grafen stelt GASEL in staat om beter te presteren dan bestaande analyses. We passen GASEL toe in een grootschalig empirisch onderzoek naar beveiligingsgeuren in open-source Ansible-code.

De vierde bijdrage is een empirische studie van de softwaretoeleveringsketen van Ansible-software. We stellen een geautomatiseerde analyse voor die

afhankelijkheden van derden van Ansible-plugins identificeert en passen deze toe om de soorten afhankelijkheden te bestuderen die in de praktijk voorkomen.

De empirische studies die in deze thesis worden gepresenteerd, vragen om verbeteringen in de kwaliteitsborgingspraktijken van Infrastructure as Code, terwijl geavanceerde tooling hiervoor wordt mogelijk gemaakt door de voorgestelde statische analyses.

Acknowledgements

First and foremost, I would like to thank my promotor, Coen De Roover, for his guidance during the course of my PhD. Coen provided tremendous support that helped shape this work, and created numerous opportunities that furthered my academic career. Although I sometimes may have been hesitant to take these opportunities—most of all, the opportunity of doing a PhD itself—Coen sunk countless hours into motivating and convincing me, without which this work would not have been possible. Coen, thank you for your guidance during these past five years.

I would also like to express my gratitude the academic jury members: Profs. Drs. João F. Ferreira, Dimitris Mitropoulos, Viviane Jonckers, Antonio Paolillo, Beat Signer, and Kris Steenhaut. Their expertise and the insightful discussions during the private defence provided valuable feedback that helped improve this text substantially.

Thank you to my co-authors, whose help in writing the papers that make up this dissertation cannot go understated. In particular, I would like to thank Ahmed for providing support with the statistical analyses and the design of the empirical studies; and Bram Adams, for hosting me during my research visit in Canada and helping shape what became the final chapter of my PhD.

I also want to thank all my colleagues at the SOFT lab for the numerous interesting discussions and fun moments, both during and after office hours. Although I cannot thank all of you individually, I'd like to highlight a few people in particular: Ahmed, for teaching me the difference between the Arabic sounds of "kha" and "kha"; Bram, for the *welcome* critical discussions, and for the countless "late-night keynotes" we attended together at VISSOFT; the Jenses, for reminding us of the importance of having lunch on time, but not a second too early; and Noah, for making the train commutes less boring—grazie!

Ik wil ook mijn ouders en "plus-ouders" bedanken voor de onvoorwaardelijke steun doorheen al deze jaren, en mij alle mogelijke kansen te geven die mij toelieten om hier vandaag te staan. And finally, thank you to my partner, Leisha, for her endless support, both during the good and the difficult times; and our two special little ladies, Boo and Bobbin, for motivating me to persevere so that I can continue to buy them their premium cat food.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
CFG	Control-Flow Graph
CPG	Code Property Graph
IaC	Infrastructure as Code
OS	Operating System
PDG	Program Dependence Graph
RQ	Research Question
SBOM	Software Bill of Materials
SCA	Software Composition Analysis

Chapter 1

Introduction

Deploying software is a complex endeavour. For instance, deploying a simple web service already requires substantial effort. First, practitioners need to acquire a server to host the web service. Moreover, they must install and configure the software's environment, including programming language runtimes, database software, a firewall, and other operating system packages. They then need to install the software itself, and configure it appropriately. For instance, they need to provide the software with the database's port number and user credentials such that a database connection can be established. Similarly, the firewall needs to be configured to only allow external traffic to the application layer and to prevent direct access to the database.

As the software grows more complex, so does its deployment process. To handle increased load, the database may need to be moved to a separate server. Similarly, the service may get replicated across several identical servers, each requiring the same deployment process. This then requires deploying a load balancer that distributes the load across these replicas. Developers may also want an isolated instance of the software to test their latest changes on, again replicating the entire deployment process. They may also want to install and configure monitoring software to gather resource usage analytics, backup software for disaster recovery, etc. Soon enough, what once was a simple web service will require tens to hundreds of individual environments and machines. This intricately interconnected network of machines makes up the *digital infrastructure* that supports the software.

Clearly, managing this infrastructure by hand has become infeasible. There are too many machines and environments, and the likelihood of human error is high. Instead, automation is desired. This automation allows the practitioners to quickly add servers to the infrastructure and set them up to be identical to existing ones. It also supports changing the existing environments in the infrastructure, or create variants for testing and development purposes. To create this automation, the practitioners write executable code scripts, akin to the source code they wrote for the software they are deploying. In short, they apply the practice of *Infrastructure as Code* (IaC).

Infrastructure as Code offers many benefits beside automation. Codifying the infrastructure processes reifies them to be documented and auditable [78], and allows the processes to be modularised, reused, and parametrised [67].

The infrastructure code can be maintained alongside the application code and included into its version control repository, allowing them to co-evolve [60]. The same well-established software engineering practices traditionally applied to the software's source code can now be applied to the infrastructure code, including testing [45], debugging, reviewing, refactoring, optimising, etc.

Nonetheless, Infrastructure as Code is no silver bullet. Although automation helps prevent human error, the automation code may itself contain flaws. Testing IaC is difficult due to the complexity of IaC languages and the environments they configure [40], causing recurring flaws in infrastructure tests [154]. Moreover, developers are used to sophisticated development tooling that supports them in assuring the quality of their software code. For Infrastructure-as-Code artefacts, tools to support developers are either overly simplistic or entirely non-existent.

For instance, in general-purpose programming languages, **code linters and bug detectors** enable developers to fix errors early [5]. Although research has suggested ways to improve the quality of Infrastructure-as-Code projects [110], IaC is plagued by defects [102], the likelihood of which is increased by certain development antipatterns [103]. Moreover, studies of Q&A forums has shown that developers have difficulties understanding some aspects of IaC languages [7, 107].

Similarly, **automated security testing** can aid developers in preventing security weaknesses and vulnerabilities from being exploited [75]. For Infrastructure as Code, research has attempted to understand practitioners' perceptions of security alerts [51, 106, 108] and has investigated means to improve security awareness [35], yet security issues are plentiful [4] and existing security scanners are inaccurate [113].

Finally, **software composition analysis** tools can aid developers in securing their software supply chain to prevent them from being compromised and causing widespread havoc [84]. Although existing research has attempted to understand the components in IaC projects [10], IaC software dependencies remain understudied, inhibiting our understanding of deployment software supply chains.

1.1 Problem Statement

We have seen that Infrastructure-as-Code files can be plagued by several types of issues. However, developers lack advanced tooling to identify such issues in their projects. This dissertation envisions sophisticated tool support for infrastructure development activities, aiding developers in quality assurance of their infrastructure code. We formulate the problem statement of this thesis as follows.

> *There is a need for sophisticated, domain-specific software quality assurance approaches for Infrastructure as Code.*

1.1.1 Motivating Example

Listing 1.1 illustrates this problem statement using a contrived example script written in the Ansible IaC language. The script performs a simplified version of the deployment process described earlier. It is parametrised with two variables (lines 3–4), containing the database credentials consisting of a name and a password. The password is initialised with an expression, demarcated by double braces, that generates a random string of words. The script then performs six "tasks" that carry out the steps in the deployment process on each targeted machine. It first installs an operating system package for the NodeJS programming language runtime (lines 6–9), and then similarly installs an operating system package for the database engine (lines 11–14). Afterwards, it configures the database's user credentials by referring to the previously-defined variables (lines 16–20), and starts the database engine via a daemon service (lines 22–26). Subsequently, it installs the application software by downloading the packaged source code from the Internet and unarchiving it (lines 28–32). Finally, it configures the application by copying a configuration file into place (lines 34–39). This configuration file is rendered from a template that is parametrised with the database credential variables from lines 3–4.

However, this script contains several flaws. First, due to Ansible's expression evaluation semantics, which we describe in detail in Chapter 2, the expression on line 4 is evaluated anew for each variable dereference. As it generates random values, the password used in the database configuration (line 19) will be different from the one used to configure the application (line 39), constituting a defect. Second, the script downloads source code from the Internet (lines 28–32) but does not check the integrity of the downloaded file. This constitutes a security weakness, as malicious modifications to the remote source code would go unnoticed, enabling a software supply chain attack.

1.1.2 Challenges

We posit that advanced software quality assurance approaches could aid in detecting and alleviating recurring flaws, such as the ones illustrated above. However, we argue that infrastructure code differs fundamentally from application code, hindering the applicability of existing approaches. Specifically, infrastructure code's main purpose is to specify and manipulate the configuration data for the infrastructure. As shown in Listing 1.1, infrastructure code implements very little logic. These differences cause several challenges that need to be addressed to obtain sophisticated quality assurance approaches:

- First, the **semantics adopted by Infrastructure-as-Code languages may substantially differ from those of general-purpose programming languages** [39]. For instance, the aforementioned expression evaluation semantics employed by Ansible are unlike those found in general-purpose languages. Therefore, analysis techniques that have been shown to be effective for application code may be inapplicable to infrastructure code.

```yaml
- hosts: web-service
  vars:
    db_user: app
    db_password: "{{ lookup('random_words') }}"
  tasks:
    - name: Install NodeJS
      apt:
        name: nodejs
        state: present

    - name: Install database
      apt:
        name: postgresql-server
        state: present

    - name: Set up database user
      postgresql_user:
        name: "{{ db_user }}"
        password: "{{ db_password }}"
        state: present

    - name: Start and enable the database engine
      service:
        name: postgresql
        state: started
        enabled: true

    - name: Deploy application software
      unarchive:
        src: https://my.apps.com/my/app.zip
        dest: /app/
        remote_src: true

    - name: Configure application
      copy:
        dest: /etc/app.conf
        content: |
            db_user={{ db_user }}
            db_password={{ db_password }}
```

Listing 1.1: Example of an Ansible script that configures an infrastructure for a web service.

- Second, **the typical bugs and defects in infrastructure code differ from those of application code**. The most common defects in IaC files relate to the configuration data [102], such as the different password values illustrated above, rather than incorrect algorithms or faulty assignments [19]. Therefore, an approach that detects defects in IaC files needs to account for those domain-specific defect categories.
- Third, similarly to defects, **the typical security weaknesses in infrastructure code differ from those found in application code**. While vulnerabilities in application code often stem from misuse of low-level features such as pointers and memory buffers, such features are unavailable in IaC languages. Instead, security weaknesses in infrastructure code are mostly caused by incorrect or poorly-chosen data values, such as weak passwords, or missing configuration steps, such as missing firewall rules or missing integrity checks [106].
- Fourth, IaC projects can be divided into two layers, namely an application layer containing the infrastructure specifications, and a runtime layer containing *configuration units* that interact with targeted platforms to align their state with the desired state [32]. IaC quality assurance approaches may need to **account for cross-layer interactions between the application and the runtime layers**. For instance, application-layer infrastructure specifications are typically interpreted on central controller machines, whereas the configuration units may be executed on the targeted platforms instead. Similarly, application-layer infrastructure specifications are developed in domain-specific languages, whereas configuration units are often implemented in general-purpose programming languages (e.g., Python and Ruby).

1.2 Overview of the Approach

To address the problem statement above, this dissertation presents three approaches to software quality assurance of Infrastructure-as-Code projects written in Ansible, one of the most popular and versatile IaC tools. Each approach targets a different aspect of software quality. The first approach aims to support developers in detecting bugs and defects in Ansible code by identifying bad coding practices related to variable usages (Chapter 4). The second approach identifies security weaknesses in Ansible infrastructure code (Chapter 5). Finally, the third approach identifies dependencies on third-party software in Ansible infrastructure implementations (Chapter 6).

Each of the approaches transposes concepts and analyses from general-purpose programming languages to the Ansible Infrastructure-as-Code domain. We address the challenges mentioned above by integrating domain-specific knowledge and analyses in each approach. We account for the differences in program semantics by designing a tailored static data-flow analysis. This analysis is used to build a graph-based representation of infrastructure code (Section 3.3).

The representation is then used in the first and second approaches to perform the detection.

To show the feasibility of our approaches, we instantiate them in prototypical implementations. We apply these tools in several large-scale empirical studies to gain a better understanding of the quality of Ansible infrastructure code at large.

1.3 Contributions

This thesis makes the following main contributions:

Program Dependence Graph for Ansible We design and implement a Program Dependence Graph (PDG) representation for Ansible, a novel static representation for Infrastructure as Code. It succinctly models the control flow and data flow within Ansible code and can be used to implement sophisticated development tooling (Section 3.3). Moreover, we describe a static data-flow analysis to build this representation, accounting for Ansible's complex semantics.

Sophisticated smell detection approaches for Ansible We propose, implement, and evaluate two code smell detection approaches for Ansible based on the PDG representation. The first detects variable-related code smells to identify possible defects (Sections 4.2 and 4.3), while the second detects security smells to identify possible weaknesses (Section 5.3). These improve upon the state-of-the-art detection approaches by incorporating semantic information from the graph representation.

Uncovering software supply chains of Ansible deployments We describe a taxonomy of the types of software that can occur in Ansible Infrastructure-as-Code deployment software supply chains (Section 6.2). Moreover, we provide a catalogue of 5 dependency management practices implemented by Ansible plugins. These are constructed from a manual review of 266 documented requirements.

Software Composition Analysis for Ansible We propose, implement, and evaluate a Software Composition Analysis that identifies the dependency management practices described in the previous contribution (Section 6.3). This analysis can automatically identify an Ansible plugin's dependencies with high accuracy. It forms a stepping stone towards automated Software Bill of Materials (SBOM) generation for IaC.

Large-scale empirical studies into Ansible code quality We conduct three empirical studies in which we apply our approaches on a large scale to quantitatively investigate Ansible infrastructure code quality in practice. The first investigates the prevalence and the lifetime of variable-related code smells in 21 931 open-source Ansible roles (Section 4.4). The second investigates the effects of semantics within security smells in a dataset of over 15 000 Ansible scripts (Section 5.4). The final study investigates

Ansible deployment supply chains to identify the prevalence of different types of software (Section 6.3) across 187 Ansible collections.

1.4 Publications

This section lists the international peer-reviewed publications made over the course of this PhD research.

1.4.1 Supporting Publications

This dissertation is supported by 4 papers published in the proceedings of international, peer-reviewed conferences, 1 journal article, and 1 chapter in a peer-reviewed book. Their relevance to the dissertation is described below.

Unless noted otherwise, in each of the following publications, I designed, implemented, and evaluated the technical contributions, designed and conducted the empirical analyses, and wrote and revised the manuscripts. Ahmed Zerouali contributed to the statistical analyses in the empirical studies. Both Coen De Roover and Ahmed Zerouali contributed to the design of the conducted experiments, and reviewed and edited the manuscripts.

- Ruben Opdebeeck, Ahmed Zerouali, Camilo Velázquez-Rodríguez, and Coen De Roover. "Does Infrastructure as Code Adhere to Semantic Versioning? An Analysis of Ansible Role Evolution". In: *Proceedings of the 20th IEEE International Working Conference on Source Code Analysis and Manipulation*. SCAM 2020 (Virtual Event, Adelaide, Australia, Sept. 27–28, 2020). Ed. by Foutse Khomh, Cristina Cifuentes, and Nikolaos Tsantalis. Los Alamitos, CA, USA: IEEE, 2020, pp. 238–248. ISBN: 978-1-7281-9248-2. DOI: 10.1109/SCAM51674.2020.00032

- Ruben Opdebeeck, Ahmed Zerouali, Camilo Velázquez-Rodríguez, and Coen De Roover. "On the practice of semantic versioning for Ansible Galaxy roles: An empirical study and a change classification model". In: *Journal of Systems and Software* 182, 111059 (Dec. 2021): *Special Section on Source Code Analysis and Manipulation*. Ed. by Wing-Kwong Chan, Tsantalis Nikolaos, and Cristina Cifuentes, 21 pp. ISSN: 0164-1212. DOI: 10.1016/j.jss.2021.111059

This conference paper and its journal extension empirically investigate the release versioning of reusable Ansible code. Although the majority of this research does not support this dissertation directly, these papers introduce the structural model (Section 3.2) and lay the groundwork for the dataset used in two of the empirical studies (Sections 4.4 and 5.4).

- Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. "Andromeda: A Dataset of Ansible Galaxy Roles and Their Evolution". In: *Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories*. MSR 2021 (Virtual Event, Madrid, Spain, May 17–19, 2021). Ed. by Gregorio Robles, Kelly Blincoe, and Meiyappan Nagappan. Los Alamitos,

CA, USA: IEEE, 2021, pp. 580–584. ISBN: 978-1-7281-8710-5. DOI: 10.1109/MSR52588.2021.00078

This data showcase paper describes a dataset of Ansible content extracted from the Ansible Galaxy index. The presented dataset is used in the empirical studies described in Sections 4.4 and 5.4.

- Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. "Smelly Variables in Ansible Infrastructure Code: Detection, Prevalence, and Lifetime". In: *Proceedings of the 19th IEEE/ACM International Conference on Mining Software Repositories*. MSR 2022 (Pittsburgh, PA, USA, May 23–24, 2022). Ed. by David Lo, Shane McIntosh, and Nicole Novielli. New York, NY, USA: ACM, 2022, pp. 61–72. ISBN: 978-1-4503-9303-4. DOI: 10.1145/3524842.3527964

This paper presents the Program Dependence Graph representation and describes the procedure to construct it (cf. Section 3.3). Afterwards, it describes a catalogue of variable-related code smells and proposes an approach based on the graph representation to detect these smells. This detection approach is then used to perform an empirical study of variable-related code smells in practice. The paper forms the foundation of Chapter 4.

- Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. "Control and Data Flow in Security Smell Detection for Infrastructure as Code: Is It Worth the Effort?" In: *Proceedings of the 20th IEEE/ACM International Conference on Mining Software Repositories*. MSR 2023 (Melbourne, Australia, May 15–16, 2023). Ed. by Emad Shihab, Patanamon Thongtanunam, and Bogdan Vasilescu. Los Alamitos, CA, USA: IEEE, 2023, pp. 534–545. ISBN: 979-8-3503-1184-6. DOI: 10.1109/MSR59073.2023.00079

This paper presents extensions to the Program Dependence Graph representation. Then, it describes graph queries on the graph to detect security weaknesses while taking control-flow and data-flow information into account. It motivates the need to integrate this semantic information into security testing for infrastructure code through an empirical study. The paper is the basis for Chapter 5.

- Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. "Infrastructure-as-Code Ecosystems". In: *Software Ecosystems: Tooling and Analytics*. Ed. by Tom Mens, Coen De Roover, and Anthony Cleve. Berlin, Germany: Springer, 2023, pp. 215–245. ISBN: 978-3-031-36060-2. DOI: 10.1007/978-3-031-36060-2_9

This book chapter provides an introduction to the Docker Hub and Ansible Galaxy software ecosystems and describes methods to analyse them. The latter half of this chapter, which describes Ansible Galaxy, provides the basis for parts of Chapter 2.

1.4.2 Other Publications

Over the course of the PhD research, 3 additional international, peer-reviewed conference papers have been published. Although not directly supporting the work presented in this dissertation, the topics are tangentially related and inspired or were inspired by the presented work. They are described below.

- Ruben Opdebeeck, Johan Fabry, Tim Molderez, Jonas De Bleser, and Coen De Roover. "Mining for Graph-Based Library Usage Patterns in COBOL Systems". In: *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering*. SANER 2021 (Virtual Event, Honolulu, HI, USA, Mar. 9–12, 2021). Ed. by Rick Kazman, Yuanfang Cai, and Marouane Kessentini. Los Alamitos, CA, USA: IEEE, 2021, pp. 595–599. ISBN: 978-1-7281-9630-5. DOI: 10.1109/SANER50967.2021.00072

This paper describes a graph-based representation of COBOL code and a means to mine these graphs for patterns. The graph patterns can be explored to find opportunities to modernise legacy code. This graph representation partially inspired the Program Dependence Graph for Ansible proposed in Section 3.3.

- Ahmed Zerouali, Ruben Opdebeeck, and Coen De Roover. "Helm Charts for Kubernetes Applications: Evolution, Outdatedness and Security Risks". In: *Proceedings of the 20th IEEE/ACM International Conference on Mining Software Repositories*. MSR 2023 (Melbourne, Australia, May 15–16, 2023). Ed. by Emad Shihab, Patanamon Thongtanunam, and Bogdan Vasilescu. Los Alamitos, CA, USA: IEEE, 2023, pp. 523–533. ISBN: 979-8-3503-1184-6. DOI: 10.1109/MSR59073.2023.00078

This paper presents an empirical study of Infrastructure-as-Code projects in the Artifact Hub ecosystem. The study investigates several aspects, including licensing, security, and outdatedness. Parts of the study were inspired by prior studies that support this dissertation and the experience they provided.

- Ruben Opdebeeck, Jonas Lesy, Ahmed Zerouali, and Coen De Roover. "The Docker Hub Image Inheritance Network: Construction and Empirical Insights". In: *Proceedings of the 23rd IEEE International Working Conference on Source Code Analysis and Manipulation*. SCAM 2023 (Bogotá, Colombia, Oct. 2–3, 2023). Ed. by Leon Moonen, Christian D. Newman, and Alessandra Gorla. Los Alamitos, CA, USA: IEEE, 2023, pp. 198–208. ISBN: 979-8-3503-0506-7. DOI: 10.1109/SCAM59687.2023.00029

This paper presents an empirical study of inheritance between Docker images found in the Docker Hub ecosystem. Similar to the previous paper, parts of this study were inspired by work that supports this dissertation.

1.5 Outline

This remainder of this dissertation is structured as follows.

Chapter 2: Background and State of the Art describes the practice of Infrastructure as Code in detail and Ansible in particular. It also reviews the state-of-the-art in academic literature on quality assurance for Infrastructure as Code and identifies knowledge gaps.

Chapter 3: Representing Ansible Infrastructure Code Artefacts presents novel representations for Ansible code, namely the structural representation and the Program Dependence Graph (PDG). It describes both their structure and how they are built.

Chapter 4: Detecting Ansible Code Smells describes our catalogue of six variable-related code smells. It then proposes and evaluates an automated approach to detecting these smells in PDGs. Finally, it describes a large-scale empirical study of the code smells' prevalences and lifetimes in a large dataset of Ansible projects.

Chapter 5: Detecting Security Weaknesses in Ansible Artefacts proposes and evaluates our security smell detection approach that leverages the PDG representation to include semantic information in the detection. We also conduct a large-scale empirical study to identify the extent to which semantic information is necessary to detect security weaknesses.

Chapter 6: Software Composition Analysis for Ansible presents a qualitative empirical study into the dependencies of Ansible plugins. It uncovers the types of software contained in deployment software supply chains. We propose an automated Software Composition Analysis leveraging implementation patterns to identify a plugin's dependencies. Using this analysis, we conduct a quantitative study of deployment supply chains at a large scale.

Chapter 7: Conclusion concludes the dissertation with a summary and a discussion of opportunities for future work.

Chapter 2

Background and State of the Art

This chapter introduces the background supporting the topics discussed in the remainder of this dissertation. First, we introduce Infrastructure as Code (IaC) in Section 2.1 and discuss its fundamental principles. We also outline the different types of IaC languages and the activities they support. Subsequently, in Section 2.2, we delve into Ansible, a versatile IaC language that is among the most popular today [40, 129, 130], and explain its most important concepts by example. Then, Section 2.3 reviews the academic literature on state-of-the-art approaches for quality assurance of Infrastructure as Code. Finally, Section 2.4 concludes with a summary.

2.1 Infrastructure as Code

Nowadays, *digital infrastructures* routinely comprise hundreds of globally distributed machines that perform computations, provide storage space, or route network traffic. Moreover, infrastructure transcends the mere notion of physical computing machinery, and can also involve combinations of managed cloud services such as Infrastructure as a Service (IaaS), i.e., computing, storage, and network resources; Platform as a Service (PaaS), i.e., managed platforms, such as container orchestration platforms; or Software as a Service (SaaS), i.e., applications, such as web or database hosting, managed entirely by a cloud provider [136].

Infrastructure as Code (IaC) has emerged as an approach to facilitate the management of such complex infrastructures. It posits that managing infrastructures ought to be codified and automated, and transposes DevOps practices to infrastructure engineering [78, 136]. This enables infrastructures to be more scalable, reliable, consistent, and maintainable—promises which we shall return to in Section 2.1.1. IaC considers infrastructure similar to software and encourages practitioners to apply established principles from Software Engineering to their infrastructure definitions [78]. This includes versioning their infrastructure definitions in Version Control Systems, automatically testing the infrastructure, and adopting Continuous Integration [34] and Continuous Delivery [54] practices.

This section explores Infrastructure as Code in more depth. First, Section 2.1.1 summarises the main principles that lay the foundations of IaC and explores

the promises and advantages they offer. Subsequently, in Section 2.1.2, we study the four main activities involved in operating a digital infrastructure.

2.1.1 Principles and Promises of Infrastructure as Code

For Infrastructure as Code to provide benefits, it must adhere to a number of principles, of which the most important are sometimes referred to as the *RICE principles*: *reproducibility*, *idempotence*, *composability*, and *evolvability* [136].

First, the elements of an infrastructure must be easily *reproducible* [78, 136]. This requires processes to be *repeatable*, i.e., automated as much as possible [78], and systems to be *consistent*, i.e., two systems built from the same infrastructure scripts must behave the same [78]. Second, infrastructure automation must be *idempotent*, meaning that running the automation again on a machine that is already correctly configured should not cause any changes [136]. Third, infrastructure elements should be *composable*, i.e., it should be possible to assemble several resources but maintain each one independently. Finally, infrastructure should be *evolvable* and open to modification to accommodate frequent changes in design and requirements [78, 136].

Adhering to these principles leads to numerous advantages offered by Infrastructure as Code. First, automating the process of managing a digital infrastructure may improve its reliability [136] and resilience, since infrastructure elements are disposable and can be reproduced effortlessly [76, 78]. Moreover, automation enables scalability, as automation can be applied to 10 or 1000 machines equally, and elasticity, as automation eliminates most need for manual intervention [136]. Furthermore, documenting the processes in source code enables knowledge sharing among engineers [136], traceability through Version Control Systems [14, 78], and auditing and validation with code reviews [14, 76]. Finally, codifying infrastructure processes into composable parts enables reusing previous efforts [14], whereas openness to change improves the maintainability of the infrastructure.

2.1.2 Types of Infrastructure as Code

Just like software development encompasses planning, coding, building, and testing, operating a digital infrastructure involves several main activities. First, the infrastructure elements need to be *provisioned*, after which they need to be *configured* or instantiated from a *template*, and finally, the ensemble of individual elements needs to be *orchestrated* into a cohesive infrastructure [14].

Provisioning[1] involves creating and managing infrastructure resources that originate from cloud providers. This not only includes computing resources from Infrastructure-as-a-Service (IaaS) providers, but can also involve Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS) resources like databases, load balancers, etc. It is the provisioning tool's responsibility to interact

[1] "Provisioning" is an overloaded term in IaC literature, and often also includes configuring or deploying software on the provisioned resources (e.g., [78]). In this dissertation, we use a narrower definition to improve the distinction with other activities and tools.

with the provider to ensure that the requested resources are created. They typically also delegate to a later step in the process, such as configuration management or template deployment, once the resources are available. Due to their coupling to public cloud providers, many provisioning tools are specific to a single provider (e.g., Amazon's AWS CloudFormation, Microsoft Azure's Bicep, etc.). Nonetheless, several tools support multiple providers, such as Terraform, Pulumi, or OpenStack Heat.

After provisioning machines, practitioners can use *configuration management* to install, configure, and manage software on the machines. For instance, they may need to tweak the operating system's settings, install various software packages, and configure appropriate firewall rules. Although these actions could be scripted in shell programming languages, configuration management tools distinguish themselves by providing domain-specific languages to configure many machines in parallel. Moreover, these domain-specific languages offer tailored abstractions to manage specific aspects of a machine's configuration. Prominent configuration management tools include Ansible, Chef, Puppet, Salt, and CFEngine, which differ in various ways. For instance, Puppet, Salt, and CFEngine enable practitioners to state the desired state of a machine declaratively whereas Ansible and Chef adopt a more procedural style. Similarly, Chef and Puppet adopt a pull-based approach, in which an agent is installed on the machine and pulls the configuration specifications from a central manager. Conversely, Ansible adopts a push-based approach where the central manager pushes the required changes to the managed machines. Specifically, Ansible uploads temporary executable scripts to each managed host which, when executed, carry out the required changes. By using standard protocols, such as Python scripts uploaded via an SSH connection, this can avoid the need for omnipresent agent software.

Rather than installing and configuring software on each machine individually, *infrastructure templating* tools allow practitioners to create a machine image as a template from which the machine can be booted. Tools such as Packer and Vagrant create virtual machine images which virtualise all aspects of the host machine, including hardware. This provides high levels of isolation, but may introduce significant overhead. In contrast, tools such as Docker and Podman create container images, which reuse the host machine's hardware and operating system and only isolate user-level processes. They are more lightweight than virtual machines, but also provide less security guarantees. Both types of images enable the *immutable infrastructure* paradigm, in which configurations cannot be modified except by redeploying the entire image, as an alternative to *mutable infrastructures* resulting from configuration management.

Finally, *orchestration* tools manage ensembles of infrastructure machines. They can provide service discovery mechanisms to enable communication between different services in the infrastructure, and handle incoming network traffic to apply load balancing and autoscaling according to demand. Moreover, they can automate the deployment of server and container images on provisioned machines. Examples of orchestration tools include Kubernetes and Docker Swarm for container images, or Nomad for virtual machine images.

This overview shows that different activities are supported by different sets of tools; therefore, practitioners typically use a combination of various tools [40]. For instance, one may use Terraform as a provisioning tool together with Packer for server templating, or Docker and Kubernetes as container image and orchestration tools, respectively. Moreover, although infrastructure templating and configuration management appear to be mutually exclusive, one can use a tool like Ansible to specify the contents of an image generated by Vagrant.

2.2 The Ansible Automation Language

Ansible is an IT automation engine whose primary purpose is configuration management, but can also be used for provisioning and orchestration [76]. Due to Ansible's simplicity and versatility, it has become a popular tool to apply Infrastructure-as-Code practices [40, 129, 130]. Ansible practitioners write scripts, called *playbooks*, that specify the infrastructure automation steps. These playbooks are executed on a machine called the *controller* (such as a practitioner's workstation), and configure a collection of remote machines referred to as *hosts*.

At its core, Ansible is an interpreter for a domain-specific configuration language based on YAML. This section reviews Ansible's most important concepts, starting with the language's basic building block, the *task*, in Section 2.2.1. Section 2.2.2 proceeds by exploring how tasks can be organised using *plays* and *playbooks*, after which Section 2.2.3 describes how tasks can be reused using *roles*. Subsequently, we detail variables, expressions, and the intricacies of their semantics in Section 2.2.4. Afterwards, Section 2.2.5 briefly describes Ansible's extensibility through modules and plugins. Finally, Section 2.2.6 describes *Galaxy*, Ansible's online registry of reusable third-party content. The terminology presented here is summarised in Table 2.1, found at the end of this section.

2.2.1 Tasks

Tasks are the fundamental building blocks of Ansible, each describing a step in a configuration process. Tasks are usually declarative, specifying a part of the desired state of an infrastructure. For instance, a task could specify that a user with a certain name should exist on a machine; that certain directories need to exist on a file system; or that some package should be installed in case the host is running a certain operating system. Importantly, practitioners do not specify *how* the task must achieve this, instead delegating those details to Ansible's underlying implementation. This makes the task a powerful abstraction, as the underlying implementation may provide properties that are desirable for IaC, such as portability across platforms.

Ansible executes each task by applying the necessary changes over a network connection in an agentless, push-based manner. It executes a single task on all remote host machines in parallel. When provided a list of tasks, Ansible executes each task sequentially, waiting for all hosts to finish a preceding

```yaml
name: Ensure user exists
user:
    name: ruben
    group: staff
    state: present
```

(A) Simple task.

```yaml
name: Ensure directories exist
file:
    path: '{{ item }}'
    owner: ruben
    state: directory
loop:
    - ~/Documents
    - ~/Downloads
    - ~/Photos
```

(B) Looping task.

```yaml
name: Ensure GNU tar package is installed on macOS
homebrew: name=gnu-tar state=present
when: ansible_facts["distribution"] == "MacOSX"
```

(C) Conditional task with shorthand argument syntax.

LISTING 2.1: Examples of Ansible tasks.

task before continuing to the next. To avoid potential stalling caused by irresponsive hosts, a task's execution time can be limited using a timeout.

Listing 2.1 depicts Ansible code for the example tasks described above. As illustrated, the task's syntax takes the form of a key-value mapping, where each key represents a directive that determines the task's execution semantics. For instance, the `name` directive assigns an optional descriptive name, as exemplified on the first line of each task. In the following, we will explore a selection of these directives and their semantics.

Actions and Arguments Each task contains a single *action* which usually[2] describes the desired state of an infrastructure element. Actions can be identified as the task's directive that does not correspond to any of the predefined directives, of which there should always be exactly one. For instance, the action in Listing 2.1a is `user`, which abstracts user accounts on a machine. Similarly, Listing 2.1b uses the `file` action to manage files and directories, whereas Listing 2.1c's action is `homebrew`, which interacts with the Homebrew package manager on macOS systems.

The action's arguments are provided as the value to the action's key in the task mapping. These typically take the form of a nested mapping, such as on lines 3–5 in Listing 2.1a. The nested mapping's keys and values correspond to argument names and values, respectively. For example, Listing 2.1a specifies that a user named "`ruben`" in group "`staff`" must be present on the machine.

[2] Not all actions are declarative. Some actions, such as `command`, imperatively specify a binary command to run. Ansible also provides some meta-commands, such as `include_tasks`, which influence the way scripts are executed.

Alternatively, arguments may be specified using *shorthand syntax*, in which the argument names and values are separated by an equals sign (=) and are listed in a single string, as exemplified in Listing 2.1c, line 2. Finally, a number of actions only take a single *free-form* argument, provided directly as a value to the action's key instead of through a nested mapping.

Iteration and Loops An action can be executed iteratively using the `loop` directive, as exemplified in Listing 2.1b. Here, the `file` action will be executed for all items in the YAML list spanning lines 7–9. In each iteration, the current list element is exposed through a variable named `item`. Moreover, the `loop_control` directive can be used to influence the iteration behaviour, e.g., to change the name of the loop variable.

Conditional Tasks Tasks can be executed conditionally by specifying a predicate using the `when` directive, as depicted on line 3 of Listing 2.1c. If the predicate returns false, the task is skipped and the action is not executed. In case the `when` and `loop` directives are used together, the predicate is evaluated for every item in the list that is iterated through.

Blocks A sequence of tasks can be grouped into a *block*, which optionally supports exception handling mechanisms for failed tasks. Moreover, tasks will inherit the directives applied at the block level. Ansible evaluates these directives separately for each task, as if they were defined on each task individually, rather than once for the entire block. For instance, a block-level `when` conditional will be evaluated for each task in the block individually.

Handlers A *handler* is a special type of task that can be used to react to changes made by another task. A task can notify a handler through the `notify` directive, which registers the handler to be executed at the end of the script execution. Note that a handler cannot notify itself. If not notified, a handler is not executed. For instance, a handler could be used to restart a web server after a preceding task made a change to the server's configuration file, but keep the server running normally if the file remained unchanged.

Importing and Including Tasks When task sequences grow, maintaining them in a single YAML file becomes infeasible. Therefore, task sequences can be split across multiple files and *imported* or *included*. Two meta-actions exist to achieve this, namely `import_tasks` and `include_tasks`, respectively, each taking the name of the file as the sole free-form argument.

The difference between importing and including tasks lies in when the file is loaded. When importing, the target file is loaded and included into the task sequence at parse time. The loaded tasks behave like a block, with each task inheriting the directives defined on the original `import_tasks` task. Conversely, including tasks happens at run time, enabling the file name to be specified dynamically using variables and expressions (cf. Section 2.2.4). Moreover, directive inheritance does not apply, meaning that a `when` directive's conditional is only evaluated once, and that the task sequence can be executed in its entirety for each item in a `loop` directive's list. Finally, a dynamically-included file can recursively include itself, causing Ansible to technically be Turing-complete [86].

```
1  - hosts: database-servers
2    tasks:
3      - name: Ensure database exists
4        postgresql_db:
5          name: app-db
6          state: present
7      - name: Ensure database user exists and has access
8        postgresql_user:
9          db: app-db
10         name: app
11         priv: ALL
12         state: present
13
14 - hosts: web-servers
15   tasks:
16     - name: Install NodeJS
17       apt:
18         name: nodejs
19         state: present
20     - name: Deploy app from git repository
21       git:
22         repo: https://github.com/my/repo
23         dest: /app
```

Listing 2.2: Example of an Ansible playbook to configure database and web servers.

2.2.2 Plays and Playbooks

Tasks are assembled into *plays*, each of which targets a group of machines to be configured identically. An Ansible script, referred to as a *playbook*, may contain several such plays to orchestrate deployments of different types of machines.

Listing 2.2 exemplifies a playbook containing 2 plays (lines 1–12 and 14–23). Each play targets a group of hosts, specified using the `hosts` directive, such as the `database-servers` (line 1) and `web-servers` (line 14) groups. The tasks to be executed in a play are listed under the `tasks` directive (e.g., on line 2).

2.2.3 Roles

It is common for parts of different plays to perform similar configuration tasks. For instance, both web servers and database servers may require network interfaces to be configured and certain firewall rules to be set up. Similarly, one may want to set up a test environment with a database server that uses the same configuration as the production environment. Rather than duplicating

```yaml
- name: Ensure database exists
  postgresql_db:
    name: "{{ database_name }}"
    state: present
- name: Ensure database user exists and has access
  postgresql_user:
    db: "{{ database_name }}"
    name: "{{ user_name }}"
    priv: "{{ user_privileges }}"
    state: present
```

(A) The `postgres` role's `tasks/main.yml` file.

```yaml
database_name: database
user_name: app
user_privileges: ALL
```

(B) The `postgres` role's `defaults/main.yml` file.

```yaml
- hosts: database-servers
  roles:
    - role: postgres
      vars:
        database_name: app-db
```

(C) Playbook using the `postgres` role and overriding the `database_name` parameter.

LISTING 2.3: Example of an Ansible `postgres` role adapted from Listing 2.2.

the tasks across the different plays, it is possible to modularise and reuse them in *roles*, akin to "packages" or "libraries" in general-purpose languages.

To execute the role's task sequence, Ansible offers several mechanisms to include roles into a play. The `roles` directive on plays can be used to include roles *statically*, akin to `import_tasks`, causing the roles to be executed before the play's task sequence. A role can also be included dynamically using the `include_role` meta-action. Similarly to `include_tasks`, this enables the role name to be specified using an expression, and supports conditional or iterative execution of the role inclusion.

A role can also contain its own set of handlers, which will be executed at the end, if notified. Moreover, roles can contain sets of variables, some of which may be overridden by the play, the exact details of which will become clear in Section 2.2.4. This mechanism enables roles to be parametrisable, improving their reusability in different contexts.

Listing 2.3 depicts an example role, which is adapted from the first play in Listing 2.2. Listing 2.3a depicts the role's task list, which is similar to the one of the original playbook. The role's list of *defaults*, the default values for its parameters, is shown in Listing 2.3b. These variables are dereferenced by expressions in Listing 2.3a, demarcated by double braces, which are explored

in more detail in Section 2.2.4. Finally, Listing 2.3c exemplifies a play that uses this role and overrides the `database_name` parameter.

Role Structure Rather than a single YAML element, a role is a directory comprising several YAML files. These directories follow a predefined structure, in which elements of a certain type (e.g., tasks, variables, or handlers) are grouped into a certain subdirectory. Most subdirectories contain a `main.yml` file, which Ansible loads by default if present, while other files must be loaded using a meta-action. Each directory is optional and can be omitted in case there are no elements of the given type. The following provides an overview of the possible subdirectories and their contents.

- The `tasks` subdirectory contains files with the role's task sequences. When the role is executed, the `main.yml` file in this directory is loaded automatically.
- The `handlers` subdirectory contains the role's set of handlers. They can be notified by tasks within the role, but also by tasks from the including play.
- The `defaults` and `vars` subdirectories contain files specifying the role's variables. These differ in their precedence, with *default variables* being easy to override with new values and serving as the means for the practitioner to parametrise the role's behaviour. In contrast, *role variables* in the `vars` subdirectory have high precedence (cf. Section 2.2.4), and are often used as constants.
- The `meta/main.yml` file contains metadata for use in the Ansible Galaxy registry (cf. Section 2.2.6), such as author, description, licence, supported platforms, etc. It also lists the role's dependencies on other roles, which are executed beforehand.
- The `files` and `templates` subdirectories contain resources for the role to install on managed hosts, such as configuration files. Files in the former are copied verbatim, whereas files in the latter are templated with the Jinja2 templating language.
- The `library` subdirectory contains custom modules and plugins to extend Ansible's functionality (cf. Section 2.2.5).

2.2.4 Variables and Expressions

Ansible practitioners can use *variables* to provide names to the data in their configuration scripts. They can reference these variables in *expressions* written in the Jinja2 templating language that is embedded in Ansible. Expressions can then be passed as values for task directives, such as action arguments or to produce a list of values for the `loop` directive. Listing 2.4 depicts a task using variables and expressions, based on Listing 2.1a. The `vars` (lines 6–8) directive is used to define variables in the task's environment, where variable names

```yaml
name: Ensure user exists
user:
    name: "{{ user_name }}"
    group: "{{ user_group }}"
    state: present
vars:
    user_name: ruben
    user_group: staff
```

Listing 2.4: Task from Listing 2.1a adapted to use variables and expressions.

and initialisers are given as a key-value mapping. The argument values lines 3 and 4 exemplify expressions, demarcated by double braces.[3]

Variable Definitions Ansible supports several mechanisms to define variables. Users can define *local* variables at various places in their code, e.g., for a single task (cf. Listing 2.4), for all tasks in a block or play, etc. They can also define *global* variables, e.g., through inventory files that associate variables with each managed host. Moreover, before executing the tasks in a play, Ansible gathers information about every host, such as its IP address, OS platform, etc., and stores these values in global *facts*. Roles can define their own variables that can be reused across plays using files in the `defaults` and `vars` directories (cf. Section 2.2.3). Furthermore, like tasks, variables can be modularised into separate files and included with the `include_vars` meta-action. It defines the variables in a global environment, where they remain visible for the remainder of the play. Variables in the `vars` directive on a task using the `include_tasks` action are referred to as "include parameters" and remain defined for all tasks executed as a result of the inclusion. Finally, practitioners can define their own host facts, referred to as *non-persistent facts*, using the `set_fact` meta-action or the `register` task directive. The latter binds a variable to a data structure that represents the result of executing the task (e.g., whether execution was successful, whether changes were made, etc.).

Except for variables defined by the `set_fact` action, which eagerly evaluates initialisers, a user-defined variable defined with an initialiser expression is a name for that expression, not the value it produces. The expression is not evaluated until the variable is dereferenced, and is evaluated anew for each dereference. Moreover, the initialiser expressions do not close over the environment in which they are bound to a variable. Instead, variable references within an initialiser are resolved according to the environment in which the expression is evaluated, i.e., every environment in which the variable they are bound to is dereferenced.

[3] Line 3 of Listing 2.1c is in fact also a Jinja2 expression. However, double braces are not required for `when` directives, as Ansible expects all conditions to be expressions.

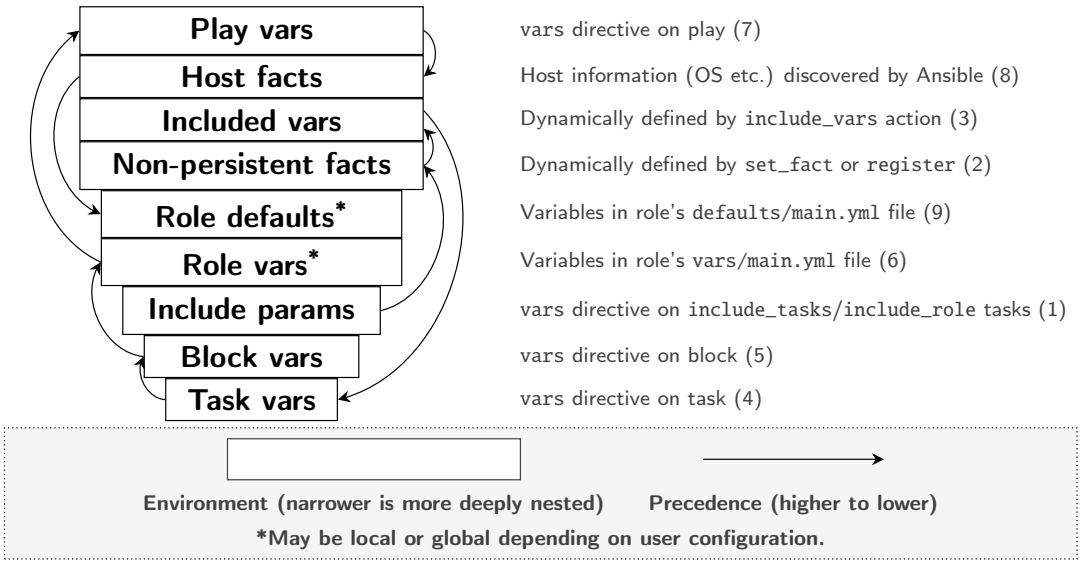

FIGURE 2.1: Summary of variable scoping and precedence rules. Narrower rectangles indicate more deeply nested environments.

Variable Scoping and Precedence Variables are bound in a certain environment. To illustrate, Figure 2.1 depicts several environments and their nesting, with lower and narrower rectangles representing more deeply nested environments. The top 4, including play variables, facts, and variables defined through `include_vars`, are global to a play, and their definitions are visible in each task executed in the play. Role defaults and role variables are defined automatically when the role is loaded and are visible for all tasks in the role. By default, these variables also remain visible in the rest of the play even after the role is left, but this can be disabled by the user of the role. Furthermore, include parameters on `include_tasks` actions are in scope for all tasks executed by the inclusion. Finally, block variables are visible to all tasks in the block, and task variables are only visible to the task itself.

In lexical scoping, a variable definition in a more deeply nested environment takes precedence over a definition of the same name in outer environments. In contrast, Ansible variable precedence is governed by 22 precedence rules,[4] the order of which does not follow the nesting of environments. Figure 2.1 uses arrows to depict these precedence rules. It shows the possibility of variable definitions in an outer environment taking precedence over variable definitions in a local environment. For instance, variables defined by the `set_fact` action (i.e., non-persistent facts) or those included via the `include_vars` action are globally visible, yet take precedence over variable definitions in local blocks or tasks.

Expressions Wrapping an expression in double braces and embedding it into a string turns the string into a *template*. Ansible evaluates the expression between double braces, possibly recursively for variable references resolving to other templates, and substitutes the result into the template string. For

[4] `https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html#understanding-variable-precedence`

instance, the result of evaluating the expression on line 3 of Listing 2.4, which merely refers to a variable, will be "`ruben`".

Expressions can be used for more than merely referring to variables. Ansible uses the Jinja2 templating language, which supports a rich Python-like expression language, including arithmetic, comparative, and logical operators; function calls; and array and property indexing. Jinja2 extends this Pythonic foundation with *filter* and *test* expressions. A *filter* expression can be used to transform data, e.g., the `user_name | upper` expression will apply the `upper` filter on the result of the left-hand side expression. *Test* expressions are similar but apply a predicate and thus always return a boolean value. For instance, `user_group is sequence` tests whether the left-hand side is a sequence. Finally, Ansible defines the *lookup* function that produces data from external sources, such as `lookup("file", "/etc/hosts")` which reads the content of external files.

2.2.5 Plugins, Modules, and Collections

Ansible's functionality is extensible through a plugin system. For instance, additional Jinja2 filters and plugins can be defined using *filter plugins* and *test plugins*, respectively. Similarly, *lookup plugins* enable defining new sources for Ansible's lookup function in Jinja2 expressions. *Inventory plugins* construct an *inventory*, a list of hosts to be configured. Finally, *modules* are special types of plugins that implement the logic behind actions executed by tasks. Whereas other plugins are executed as part of the Ansible process on the controller, modules execute on the host under configuration. Therefore, plugins have to be implemented in Python, while modules can be implemented in any language.

Earlier versions of Ansible curated these plugins in a centralised mono-repository. As the number of plugins grew, Ansible introduced *collections* of plugins and migrated the majority of plugins to dedicated collection repositories. A collection can be identified uniquely by the combination of a namespace (author) and name, usually separated by a period. A collection's content commonly shares a single theme, such as the *community.docker* collection of plugins for working with Docker, or the *amazon.aws* collection of plugins to interact with Amazon's AWS cloud.

2.2.6 The Ansible Galaxy Registry

Ansible Galaxy[5] is a registry operated by Ansible to facilitate reusing open-source Ansible content. It collects and displays metadata about open-source, reusable roles (cf. Section 2.2.3) and collections (cf. Section 2.2.5). Ansible practitioners can discover such content via Ansible Galaxy's web interface, while a command line utility can be used to install, update, and manage the content in a project. As of May 2024, Ansible Galaxy indexes over 33 700 roles by nearly 15 000 authors, the most popular of which have been downloaded

[5] `https://galaxy.ansible.com/`

Table 2.1: Overview of Ansible terminology.

Term	Meaning
Task	Specification of a single configuration step, possibly conditionally or in a loop.
Action	Step executed by a task on the target machines.
Block	List of tasks, with optional exception handling.
Handler	Task that only executes when "notified" by another task.
Play	List of tasks and blocks, implementing automation steps for a group of hosts.
Playbook	Executable Ansible script, consisting of a list of plays.
Role	Reusable abstraction that groups related tasks and variables.
Controller	Machine that runs the main Ansible interpreter.
Host	Machine managed using Ansible tasks.
Fact	Variable containing information about a host.
Inventory	List of hosts and associated facts.
Jinja2	Template expression language used by Ansible.
Template	String containing Jinja2 expressions.
Filter	Data manipulation expression in Jinja2.
Test	Predicate expression in Jinja2.
Lookup	Function for use in templates to retrieve data from external sources.
Plugin	Mechanism to extend Ansible functionality.
Module	Special type of plugin implementing the logic to execute an action.
Collection	Mechanism for 3rd-party reuse and distribution of related Ansible content.

millions of times. Moreover, it hosts nearly 3 000 collections, some with more than 70M downloads. This makes Ansible Galaxy the largest source of openly-available Ansible content.

2.3 State of the Art in Quality Assurance for Infrastructure as Code

Infrastructure as Code is an emerging research domain, with an increasing number of works published each year [104]. Because practitioners face several challenges when performing quality assurance of IaC scripts [40], many studies have focused on supporting these activities [104]. In this section, we survey the academic literature for works that propose automated IaC quality assurance approaches, such as tools and analyses. We classify the proposed approaches according to two dimensions, namely the quality aspect considered (Section 2.3.1) and the analysis techniques used (Section 2.3.2). Section 2.3.3 provides an overview of the existing approaches and identifies gaps in the literature that will be addressed in subsequent chapters.

2.3.1 Aspects

We discern three high-level quality aspects that have been central to prior academic studies.

First, **correctness** of infrastructure code forms a major concern for practitioners, and bugs and defects are plentiful [102]. Moreover, practitioners not only consider IaC to be error-prone, but also difficult to test and debug [40]. Therefore, many studies have focused on means to identify or remediate correctness issues before they cause service outages. Several studies focus on typical infrastructure defects, such as misconfigurations or erroneous logic [102]. Others investigate IaC-specific properties, such as idempotence (cf. Section 2.1.1) [44, 55, 57], determinism of declarative models [119], or the validity of deployment topologies [15].

Second, **maintainability** relates to the ease with which infrastructure code can be changed and improved, e.g., to accommodate new infrastructure requirements. Maintainability is negatively affected by the presence of so-called *smells and antipatterns* that are indicative of error-prone programming practices that may encumber future changes to the code or its design [121]. Note that smells are not necessarily defects, so the affected infrastructure code may continue to function while the negative impacts go unnoticed for long periods of time. Therefore, prior work has investigated different types of smells in IaC, the most common being *code smells*, which manifest themselves in the infrastructure code's implementation and design [116, 117, 120, 135]. Moreover, *test smells* may indicate inefficiencies in infrastructure tests [47], whereas *linguistic inconsistencies* indicate divergence between code's behaviour and its documentation [12, 13].

Third, **security & compliance** are increasingly important in the context of Infrastructure as Code. In particular, insecure coding patterns, often referred to as *security smells*, have been subjected to numerous studies [11, 22, 31, 101, 105, 106, 108, 109, 113, 115]. Researchers have also employed IaC specifications to reduce cloud resource access privileges [122, 123] and to identify vulnerabilities in cloud infrastructures [16, 69]. Furthermore, prior work has checked IaC specifications for compliance to laws, regulations, and internal requirements [36, 66].

2.3.2 Techniques

We identify 6 classes of analysis techniques used in the quality assurance approaches proposed by prior academic work, summarised in Table 2.2. All but one of these techniques are *static* approaches [18], which extract information from infrastructure scripts without executing them. The final technique is *dynamic*, which requires running the infrastructure scripts to collect the necessary information. As this requires deploying a (virtual) infrastructure, they may be more costly to apply, and may not be universally applicable.

Syntactic analysis The simplest approach to analysing infrastructure code is by inspecting a syntactic representation. Such representations can be obtained by parsing the infrastructure code into structures such as token sequences or abstract syntax trees (AST). This already captures all the considered information, making these approaches lightweight and easy to implement.

These syntactic structures can be used in several ways. One of the earliest applications was to calculate complexity metrics, often inspired by code metrics from object-oriented programming, and use them to identify code and design smells. The Puppeteer tool by Sharma et al. [120] implements this technique for Puppet, while Schwarz et al. [117] and van der Bent et al. [135] extend existing linters for Chef (Foodcritic) and Puppet (puppet-lint), respectively.

Moreover, antipatterns and security smells can be identified using syntactic pattern matching, often specified as logical rules and string predicates. This technique has frequently been applied to identify security smells, including in tools such as SLIC [106], SLAC [108], GLITCH [115, 116], InfraSecure [113], SLI-KUBE [109], and KubeHound [31]. Furthermore, the TAMA [47] tool employs syntactic pattern matching to identify test smells in Ansible testing code.

Finally, approaches can use ad hoc techniques to extract relevant information from syntactic representations and feed it to pre-existing smell detectors. For instance, SecureCode [22] extracts shell code from Ansible scripts and checks it with ShellCheck, an existing security linter for shell code.

Data-flow analysis The syntactic representations described above only comprise information on the structure of infrastructure scripts, but lack information on the relations between different code elements in those scripts. For instance, ASTs do not relate a variable reference to the definition it resolves to when evaluated. Such information can be incorporated using a data-flow analysis.

A data-flow analysis requires the script's control-flow information, usually specified in a Control-Flow Graph (CFG). The analysis then propagates variable definitions along the CFG to the usages of that definition. The established links between definitions and usages provide information that can be used in subsequent analyses. Nonetheless, the need for a CFG and an additional analysis pass makes these approaches less lightweight than mere syntactic analyses.

Data-flow analysis can be employed to identify infrastructure code elements that are affected by security smells originating from other elements. This technique is used in TaintPup [105] and TIDAL [101] for Puppet and Ansible, respectively. These approaches use a data-flow analysis to extract data dependences, i.e., relations between two infrastructure code elements (e.g., two Ansible tasks) in which one uses the data produced by the other. They then identify security smell instances using syntactic pattern matching as described above, and propagate these instances along these data dependences to identify impacted elements.

Semantic analysis Although data-flow analyses provide more information than simple syntactic analyses, they cannot model the interactions an infrastructure script may have with its environment (e.g., file system operations), or the transitions of infrastructure states caused by executing an infrastructure script. To overcome this, researchers have used formal semantics for IaC languages, such as μPuppet [39], combined with formal verification techniques, to check properties of IaC scripts.

For instance, using formal semantics, IaC scripts can be modelled as SMT (Satisfiability Modulo Theory) formulae and provided to an SMT solver. Rehearsal [119] uses this approach to check whether the script adheres to fundamental properties. Tortoise [138] also uses SMT solving to synthesise patches to update IaC specifications based on the shell commands that practitioners run on the configured machines. Moreover, Häyhä [69] uses formal semantics to model the infrastructure state transitions needed to apply an IaC specification and checks those transitions for security vulnerabilities.

Semantic analyses can provide advanced support for quality assurance activities. However, they are generally more costly than the lightweight alternatives described above. Furthermore, they may suffer from limitations that substantially hamper the approaches' practical applicability, as the formal semantics often only supports a subset of the IaC language.

Architectural analysis Whereas the approaches described above focus on the infrastructure code and its behaviour, approaches utilising architectural analysis instead use the infrastructure code to derive a *deployment model*, i.e., a high-level model of the deployed infrastructure. Such models can represent the topology of the infrastructure, the elements thereof, the services deployed onto those elements, and the relations among the services. Deployment models are typically derived from declarative IaC templates, such as from specifications written in the TOSCA[6] [9] standard, but may still require manual effort to generate [81]. These models serve as the input to infrastructure quality assurance techniques that perform architectural analysis.

Several approaches exist that perform such architectural analysis. For instance, Sommelier [15] checks TOSCA topologies against validity constraints, whereas Krieger et al. [66] check them against compliance rules. Furthermore, Ntentos et al. check deployment models for architectural smells by calculating architectural metrics [81, 83] and searching for antipatterns [82]. Kumara et al. [68] represent TOSCA models as knowledge graphs and use graph querying to identify smells. Finally, Cauli et al. [16] specify architectural models in formal logics that can be queried for security properties.

Predictive models The previous approaches require analysis designers to encode specific knowledge, such as the properties to check for defects or particular smell patterns. Predictive models instead rely on Artificial Intelligence (AI) techniques, such as Machine Learning (ML), to learn defects and smells by example. They can then be used to make predictions about previously-unseen examples, e.g., to categorise an infrastructure script as defective or not defective, referred to as *defect prediction*.

Numerous techniques have been proposed to perform defect prediction for Infrastructure as Code. Rahman and Williams use statistical learning models, first combined with text mining techniques [111], and later with syntactic properties [112]. Dalla Palma et al. apply Machine Learning techniques to

[6] TOSCA, or the "Topology and Orchestration Specification for Cloud Applications", is an OASIS standard modelling language for cloud deployments and orchestration.

complexity metrics counted from syntactic elements [23, 24, 25, 26], a technique later expanded upon by Quattrocchi and Tamburri [98] and Begoug et al. [8]. Dalla Palma et al. [27] later also apply Machine Learning techniques on metrics inspired from object-oriented programming. Moreover, we have applied Machine Learning on infrastructure code changes to predict when wrong types of version increments have been used in Ansible role updates [93, 94]. DeepIaC [13] and FindICI [12] apply Natural Language Processing to identify linguistic inconsistencies between the implementation of Ansible tasks and their natural language description. Finally, Puppet Analyzer by Chen et al. [17] clusters the syntactic changes made in defect-fixing commits to infer patterns that represent recurring defects.

Dynamic analysis The prior approaches are all *static* and refrain from executing any infrastructure code, which limits the amount and accuracy of the collected information due to the need for approximations. Instead, *dynamic analysis* executes the infrastructure code and monitors its behaviour or the state of the infrastructure. We discern three main types of dynamic analysis, namely testing, deployment-time analysis, and post-deployment analysis.

Testing involves executing an IaC script and verifying that it performed the correct actions. Shimizu et al. use the outcome of existing tests to optimise access permissions for cloud resources [122, 123]. Other researchers have instead focused on automating the testing process rather than requiring practitioners to write test cases manually. For instance, Sokolowski et al.'s testing framework ProTI combines randomly-generated inputs ("fuzz testing") in combination with automated mocking of infrastructure elements [125, 126, 128]. Similarly, Hummer et al. [55], Hanappi et al. [44], and Ikeshita et al. [57] automatically generate test cases using model-based testing.

Deployment-time analysis employs information gathered during the execution of the infrastructure code, i.e., at deployment time. Note that the infrastructure code need not be executed against real infrastructure elements, which can instead be virtualised. By monitoring the execution behaviour, approaches can extract traces of calls to system APIs ("*syscalls*") performed by the infrastructure code. Sotiropoulos et al. [127] use these traces to detect missing dependencies between parts of the scripts, whereas Dozer by Horton and Parnin [49] uses them to rewrite shell commands to IaC abstractions.

Finally, *post-deployment analysis* occurs after an infrastructure has been deployed by an IaC script. These analyses gather information from real-life deployments and combine it with information extracted from the IaC scripts. IaCMF [36] uses such information to find compliance issues, while KubeHound [31] uses it to identify smells in configuration properties.

2.3.3 Overview of Approaches and Knowledge Gaps

Table 2.2 depicts an overview of approaches to quality assurance of Infrastructure as Code presented by prior academic research, categorised according to

the two dimensions described above. Note that a single approach may target several quality aspects, and thus may occur in several columns.

The overview enables us to make various observations about the state of academic research and to identify gaps therein. First, the table shows that very few approaches use data-flow analysis, even though this technique provides more information than mere syntactic analysis, and is more lightweight and more widely applicable than semantic analysis. Second, we see that defect detection currently requires more sophisticated techniques than simple syntactic or data-flow analyses. Third, while predictive models have proved useful to predict defects and maintainability issues, they have not yet been used to predict security and compliance issues. Fourth, whereas defects and security issues have been studied in great depth, considerably less attention has been given to maintainability issues. Finally, a notably missing quality aspect is that of software supply chains and third-party dependencies. These have been studied extensively for general-purpose programming languages [72, 146, 149] and specific domains such as CI/CD pipelines [30, 85], Docker containers [88, 150], Machine Learning pipelines [52], and microservices [37]. However, these existing studies focus on runtime and development dependencies, and we find no research that investigates the *deployment dependencies* of infrastructure code.

2.4 Conclusion

This chapter introduced the necessary background for the topics that are considered in the remainder of the dissertation. To this end, we have introduced Infrastructure as Code as the practice of managing digital infrastructures through executable code. We have described the four types of Infrastructure-as-Code languages and the activities they support: provisioning, configuration management, infrastructure templating, and orchestration. We explored an Infrastructure-as-Code language that is central to this dissertation, namely Ansible, and described its central concepts, including tasks, playbooks, roles, plugins, and the semantics of variables and expressions.

Moreover, we reviewed the academic state-of-the-art approaches for quality assurance of Infrastructure as Code. We described the three main quality aspects considered: correctness, maintainability, and security and compliance. We further outlined six analysis techniques employed by these prior approaches. From this review, we observed several gaps in the academic research that we shall alleviate in the next chapters. In particular, we identified a distinct lack of data-flow analysis techniques for Infrastructure-as-Code languages. We will address this by introducing a representation that captures data-flow information in Chapter 3, which we employ to detect code smells in Chapter 4 and security weaknesses in Chapter 5. Moreover, we found that prior work has not investigated the software supply chain of Infrastructure-as-Code artefacts, a gap which we will bridge in Chapter 6 by investigating the third-party dependencies of Ansible plugins.

TABLE 2.2: Overview of Infrastructure as Code quality assurance approaches.

	Quality aspect		
Analysis technique	Correctness	Maintainability	Security & compliance
Syntactic analysis	—	Puppeteer [120] Schwarz et al. [117] van der Bent et al. [135] TAMA [47] GLITCH [116]	SLIC [106] SecureCode [22] SLAC [108] GLITCH [115] InfraSecure [113] SLI-KUBE [109] KubeHound [31]
Data-flow analysis	—	—	TaintPup [105] TIDAL [101]
Semantic analysis	Rehearsal [119] Tortoise [138]	—	Häyhä [69]
Architectural analysis	Sommelier [15]	Kumara et al. [68] Ntentos et al. [82, 83]	Kumara et al. [68] Krieger et al. [66] Ntentos et al. [81] Cauli et al. [16]
Predictive models	Rahman and Williams [111, 112] Puppet Analyzer [17] Dalla Palma et al. [23, 24, 25, 26] Quattrocchi and Tamburri [98] Begoug et al. [8]	DeepIaC [13] FindICI [12] Dalla Palma et al. [27] Opdebeeck et al. [93, 94]	—
Dynamic analysis	Hummer et al. [55] Hanappi et al. [44] Ikeshita et al. [57] Sotiropoulos et al. [127] ProTI [125, 126, 128]	Dozer [49]	Shimizu et al. [122, 123] KubeHound [31] IaCMF [36]

Chapter 3

Representing Ansible Infrastructure Code Artefacts

Before one can perform static code quality assurance, one needs to represent the code in a way that facilitates automated reasoning. Researchers have proposed countless representations over the last decades, some more complex than others, many tailored to specific programming languages, paradigms, or problem domains. Nonetheless, code representations for Infrastructure as Code are far less studied, yet are crucial to perform quality assurance.

We commence this chapter by reviewing these existing code representations in Section 3.1. When applied to Ansible, the representations exhibit severe limitations by not considering Ansible's particularities in sufficient detail. For instance, the representations rely solely on YAML parsing, and disregard Ansible-specific syntax. Moreover, Ansible's intricate data-flow semantics is often ignored or overly approximated, leading to inaccuracies. Finally, very few IaC representations capture control-flow or data-flow information, and no IaC artefact representation relates the two.

To alleviate these limitations, we introduce two novel representations of Ansible code. The first is a structural model (Section 3.2), which effectively combines YAML syntactic information with Ansible-specific knowledge. It constitutes a hierarchical representation of Ansible code, facilitating automated reasoning about its syntax and structure. Building upon the structural model, as a second representation, we introduce a Program Dependence Graph (PDG) for Ansible (Section 3.3). This graph-based representation encodes and relates control-flow and data-flow information. We describe a whole-program static analysis to build such PDGs, accounting for Ansible's data-flow semantics and capturing the control-flow interplays between different files in a project.

These representations substantially advance the state of the art in static analysis of Ansible scripts. They form the foundation of the static quality assurance approaches that will be the subject of subsequent chapters, and pave the way for advanced yet practical quality assurance tooling.

3.1 State of the Art

Prior work has proposed several approaches that perform static analysis of infrastructure code (cf. Chapter 2). Static analyses share a need to represent the code they reason about. These representations differ in complexity depending on the information needed by the analysis. To this end, the simplest representations, which we refer to as syntactic representations, merely represent the original source code as data structures that facilitate automated reasoning, such as sequences or trees. In contrast, semantic representations encode information derived from the syntactic representations, such as relationships between different constructs in the source code. In this section, we describe these two types of representations and how they have been used in prior work. Note that we do not consider a third type of representation often considered in prior work, the architectural representation used by architectural analyses (cf. Section 2.3.2), as these represent the infrastructure itself rather than the code used to manage it.

3.1.1 Syntactic Representations

Syntactic representations are simple code representations that are the result of parsing source code into a data structure that provides convenient access to the code's constructs. They are used in syntactic analyses (cf. Section 2.3.2) and thus only represent the code's syntactic information.

In their simplest form, one can transform source code into *lexical token streams* by splitting the source code based on positions of certain character classes, such as whitespace or punctuation. However, such representations do not distinguish between different program elements. This makes it difficult to perform an in-depth analysis of the code. Nonetheless, token stream representations can be used for tasks such as code smell detection, wherein a detector matches logical rules against sequences of tokens to highlight potential problems in code [106].

Parsing enables one to transform token streams into a *concrete syntax tree (CST)* or *parse tree*. This involves assigning a syntactic class to each token and assembling the tokens into a tree according to the grammar of the programming language. The internal nodes of this tree represent syntactic classes, whereas its leaf nodes represent the tokens themselves. However, these trees contain many syntactic details that, while important for correct and deterministic parsing, are irrelevant for program analysis (e.g., string quotes, punctuation, separators, etc.). One can therefore simplify a concrete syntax tree into an *abstract syntax tree (AST)*, which represents the structure of the program but elides the unnecessary lexical details found in concrete syntax trees. Its nodes represent syntactic constructs (e.g., statements and expressions of different types), with the leaf nodes representing atomic constructs (e.g., values and names). The tree's edges represent structural containment. For instance, in an abstract syntax tree, an Ansible task would appear as a node with subtrees representing the task's directives as children. ASTs substantially facilitate

code analysis. For example, counting the number of tasks that appear in an Ansible file requires little more than a simple tree traversal. Consequently, abstract syntax trees have been used in numerous applications, such as code linting [108, 113, 120] and defect prediction [24].

Nonetheless, syntax trees, both concrete and abstract, are specific to a certain IaC language, hindering the generalisability of the analyses utilising them. *Language-agnostic representations* aim to eliminate this issue by representing source code in terms of a model of common elements. Saavedra and Ferreira's [115] Intermediate Representation implements such a language-agnostic representation as a generalised syntax tree for configuration management languages such as Chef, Ansible, and Puppet. Using this representation, they design GLITCH, a polyglot approach to detect security smells, overcoming inconsistencies in prior approaches.

3.1.2 Semantic Representations

Semantic representations incorporate the program's semantics alongside the representation of the source code, through behavioural information derived by prior program analyses. Static analysis tools can then employ the already-derived information in their own analyses. This enables the use of data-flow analysis and semantic analysis (cf. Section 2.3.2) in quality assurance techniques.

Several kinds of semantic information can be considered. Control-flow and data-flow information are among the most common requirements for in-depth static analyses. *Control-flow information* describes the possible paths that a program may take, and includes information about control order, branching points, and loops. *Data-flow information* describes how and where data is defined and used in a program.

For instance, the *Structured Resource Tree* representation by Dai et al. [22] is a tree representation that closely resembles syntax trees. However, SRTs also contain some semantic information, such as data-flow dependencies between variables and expressions, and a partial execution order relationship. Dai et al. [22] employ such models to extract shell commands that are embedded into Ansible scripts, which they subsequently scan for security weaknesses.

Data-flow information can also be captured as *data dependence*, which relates two program elements wherein the first produces data that is needed by the second. Such relationships can be represented in a *data dependence graph*, whose nodes represent program elements and whose edges encode data dependences among the nodes. They can be used to propagate certain information between program elements. For instance, a security weakness caused by one program element can be traced to impacted program elements along the edges of a data dependence graph [101, 105].

Other representations focus on *execution order dependencies* between different program elements. These form a type of control-flow information that is commonly necessary when analysing Puppet programs, as Puppet employs a

non-deterministic execution order for "resources", its equivalent of Ansible's tasks. For instance, the *resource graph* introduced by Shambaugh et al. [119] represents execution order dependencies specified by the developer in a Puppet program. These resource graphs can be used to check whether executing a Puppet program leads to a deterministic outcome, even when non-deterministic ordering is used. Sotiropoulos et al. [127] similarly build execution order dependency graphs. Using system call traces obtained dynamically, they inspect the graphs to identify missing execution order dependencies between resources.

Finally, semantic representations can also model *intermediate states* of the infrastructure and the transitions between them. For instance, Hummer et al. [55] model the state of infrastructure machines according to some properties (e.g., whether a certain package is installed or a certain service is running). They then use a State Transition Graph to derive test cases to test the idempotence of Chef code. Lepiller et al. [69] build "upgrade states" that capture all possible intermediate states a cloud topology could transition through when a change is applied. They then verify the intermediate states to ensure prior security policies would not be violated during the upgrade.

3.1.3 Limitations

The aforementioned approaches exhibit certain limitations when applied to Ansible.

For syntactic models, since Ansible code is written in YAML, one can easily obtain a syntax tree by parsing the YAML file. However, merely parsing YAML files without incorporating Ansible-specific knowledge exhibits a number of shortcomings. For instance, the elements of the syntax tree are unlabelled, i.e., without context, one cannot determine whether a key-value mapping is a task, a block of tasks, a collection of variables, or a variable value. Moreover, YAML parsing does not take Ansible-specific syntax into account, such as the shorthand syntax which allows one to write an action and its arguments on a single line (cf. Section 2.2.1), or the embedded Jinja2 expressions. Such limitations burden the designer of the analysis, who has to reconstruct syntactic contexts and may need to perform additional parsing or transformations. To alleviate these limitations, Section 3.2 will introduce a novel *structural* representation for Ansible, which incorporates such particularities into a syntax tree obtained by parsing YAML files.

For semantic models, we note that existing analyses for Ansible do not properly account for Ansible's complex semantics (cf. Section 2.2.4). For instance, the construction of the Structured Resource Tree [22] relies on "define-before-use" heuristics that incorrectly approximate data flow. Similarly, the Data Dependence Graph representation [101] does not capture control-flow information and can thus not relate control flow to data flow. This renders it unable to specify how a data value influences control flow (e.g., which branch is taken in conditionals), or how control flow influences data values (e.g., conditional definition of variables). Moreover, both representations

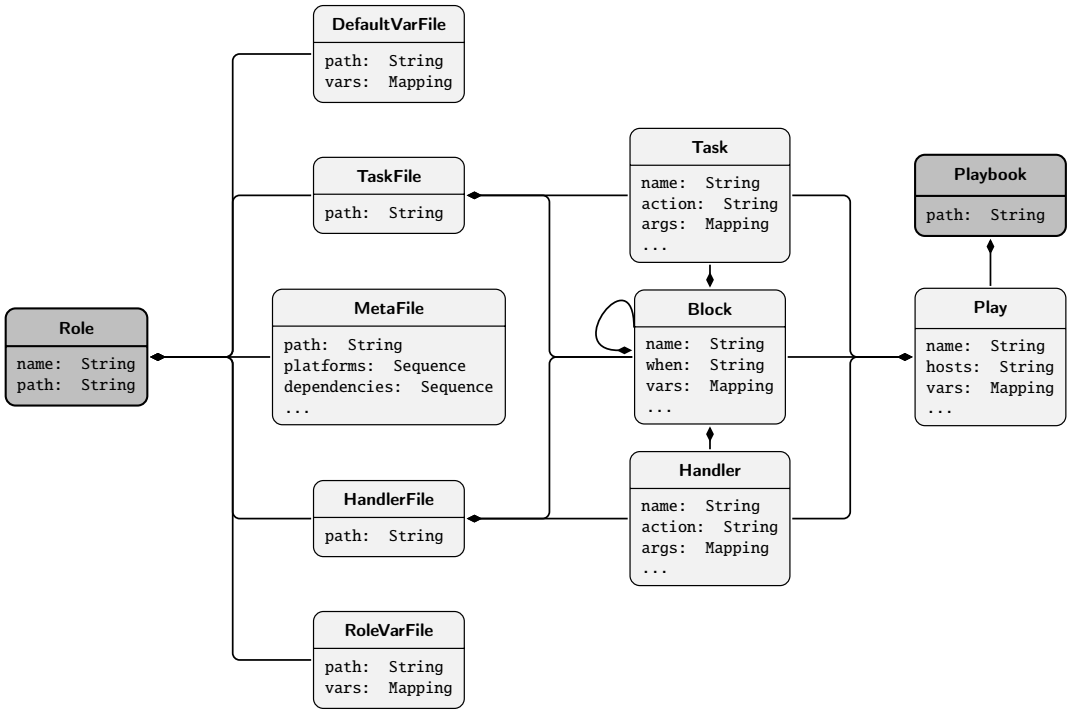

FIGURE 3.1: Schema of elements in the structural model. Elements are depicted as rectangles, with darkly-shaded rectangles representing root elements. UML composition arrows depict the hierarchy among elements. For brevity, not all possible properties are shown.

only consider a single Ansible file, although a script can include other files to consume their variables or even execute their tasks. To alleviate these limitations, Section 3.3 will introduce a novel Program Dependence Graph (PDG) representation, which combines control-flow and data-flow information, accounts for Ansible's semantics, and is built using a whole-program analysis.

3.2 Structural Model of Ansible Code

As mentioned in Section 3.1.3, existing syntactic representations exhibit several limitations caused by a lack of Ansible-specific knowledge. Therefore, in this section, we introduce a novel *structural model* for Ansible that transcends the syntactic contents of files. It forms a tree of Ansible elements, such as blocks, tasks, and variables, closely following Ansible's own internal representation of its files.

3.2.1 Structure of the Structural Model

Figure 3.1 depicts the schema of the structural model. The model constitutes a hierarchical tree that closely follows the structure of Ansible projects as described in Section 2.2. The structure of the model depends on whether the parsed Ansible project is a playbook or a role, indicated by the distinction between the two types of root nodes.

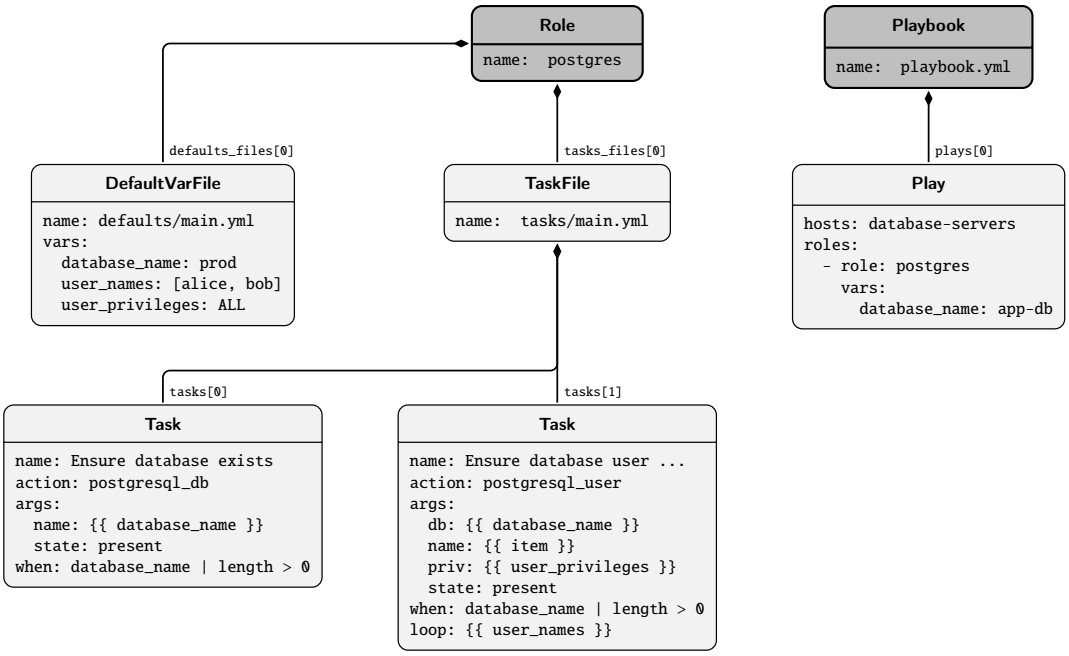

FIGURE 3.2: Structural model representation of Listing 3.1.

Playbook nodes have one or more children that represent the playbook's plays. Each play node contains a record of the play's directives, including the list of hosts it targets, a mapping of play variables, etc. The play's tasks are represented as child nodes, ordered according to the order in which they appear in the code. Like play nodes, task nodes contain a record of the task's directives, including its action, a mapping of arguments, and conditionals. The play's handlers are represented analogously, yet we make a clear distinction to emphasise the difference in control-flow semantics (cf. Section 2.2.1). Finally, when several tasks or handlers are assembled in a block, they are represented by a block node, also containing a record of its directives. As blocks can be nested, a block's child nodes can comprise other blocks. However, tasks and handlers cannot be mixed in a block's (transitive) children.

A role node's children consist of the role's files, with each child node type representing a file in either the `tasks`, `defaults`, `vars`, `handlers`, or `meta` directory. There may exist multiple files of each type, distinguished by their name, except for metadata files. The metadata file node is a singleton that represents the `meta/main.yml` file if present, and contains the file's content, such as supported platforms and the role's dependencies. Nodes representing task files have ordered sequences of task and task block nodes as children, and analogously for handler file nodes. Finally, nodes representing role variable and default variable files contain the mapping of variable names to their initialisers, as specified in the file. Although these two node types are almost identical, we make the distinction to enforce the differences in precedence (cf. Section 2.2.4).

To illustrate, Figure 3.2 depicts the structural models for the example role

```yaml
- name: Ensure database exists
  postgresql_db:
    name: "{{ database_name }}"
    state: present
  when: database_name | length > 0

- name: Ensure database user exists and has access
  postgresql_user:
    db: "{{ database_name }}"
    name: "{{ item }}"
    priv: "{{ user_privileges }}"
    state: present
  loop: "{{ user_names }}"
  when: database_name | length > 0
```

(A) The `postgres` role's `tasks/main.yml` file.

```yaml
database_name: database
user_names:
    - alice
    - bob
user_privileges: ALL
```

(B) The `postgres` role's `defaults/main.yml` file.

```yaml
- hosts: database-servers
  roles:
    - role: postgres
      vars:
        database_name: app-db
```

(C) Playbook using the `postgres` role and overriding the `database_name` parameter.

LISTING 3.1: Example of an Ansible `postgres` role, adapted from Listing 2.3.

and playbook in Listing 3.1, which is adapted[1] from Chapter 2's example (Listing 2.3). The left-hand side of the figure contains the representation for the role, with nodes for the default variable file of Listing 3.1b, the tasks file of Listing 3.1a, and its two tasks. The right-hand side depicts the structural model for Listing 3.1c's playbook and its play. One may observe that the records of directives contained in the nodes closely resemble the original YAML specifications. However, we note that the structural model goes beyond merely tagging the YAML elements, as it also normalises the representation. An example of such a normalisation can be seen in the task nodes, where the action and its arguments are separated into explicit properties.

3.2.2 Building a Structural Model

To create the structural model, we build upon the representation constructed by Ansible's internal parser, thereby benefiting from the normalisation it applies. For instance, it transforms different syntactic styles, such as the task argument shorthand syntax (cf. Section 2.2.1), into the same internal representations. Moreover, Ansible's parser identifies the task's actions and arguments, and performs type coercion of task directive values where necessary. It also performs syntax validation, enabling us to reject invalid Ansible files that a standard YAML parser would allow. Our extractor can operate in two modes, strict or lenient, which differ in how such invalid files are handled. The former aborts when a validation error is encountered, whereas the latter attempts to process as many files as possible by ignoring invalid constructs, such as malformed files or tasks with invalid or missing directives. We then post-process Ansible's internal representation by transforming its objects into the nodes of our structural model.

However, relying on Ansible's internals comes at the cost of having to work around several limitations, as Ansible's internals are engineered to *execute* Ansible code rather than statically analyse it. When parsing files, Ansible applies some optimisations that render executing the infrastructure code more efficient. For example, when Ansible encounters a task that statically imports another task file, it replaces the importing task by the content of this file. Since we aim for a structural representation, it is important to represent all task files separately, rather than inlined into another file, meaning this inlining needs to be undone. Moreover, current Ansible versions have removed support for deprecated syntactic constructs, which our extractor solves by transforming them during a pre-processing phase.

In summary, our structural model extractor leverages Ansible's internals to create a syntactic, hierarchical representation that offers a convenient means of reasoning about the structure of Ansible roles and playbooks. This representation forms a foundation for our static analyses for Ansible, which we build upon in the next section.

[1] The adaptation of example contains two changes to illustrate concepts described in this chapter, namely the addition of conditionals to both tasks, and the addition of a loop and a list variable to the second task.

3.3 Program Dependence Graph for Ansible

The semantic representations reviewed in Section 3.1.2 do not relate an IaC script's data flow to its control flow or vice versa. Moreover, existing representations that attempt to model the behaviour of Ansible code fail to account for Ansible's intricate semantics regarding template expression evaluation and variable precedence (cf. Section 2.2.4). Finally, they only analyse individual files, while Ansible scripts can include several other files, necessitating a whole-program analysis. Therefore, in this section, we introduce a *Program Dependence Graph* (PDG) [38, 95] representation for Ansible code, which captures and intertwines control-flow and data-flow information. Moreover, we describe a whole-program analysis that builds such PDGs while accounting for Ansible's semantics.

We choose a PDG-based representation not only because PDGs succinctly represent control-flow and data-flow information between program elements, but also because PDG-based representations have already proven themselves as enablers of advanced development tooling for general-purpose programming languages. Applications include optimisation [38, 96] and program slicing [95, 131, 132], code clone detection [71], refactoring [50, 97], and static security analysis [43, 62]. Moreover, derivatives of the PDG representation have been paired with graph mining algorithms to recommend code snippets [80], assess migration effort [87], mine code change patterns [79], and detect defects in library usages [3]. These numerous applications provide confidence that a PDG representation for Ansible will enable developers and researchers to build tooling for and study software engineering problems of infrastructure code.

3.3.1 Structure of the Program Dependence Graph

The Ansible PDG is a directed graph whose nodes can be categorised into control and data nodes. The former represent control-flow structures, whereas the latter represent data. These nodes are interconnected using edges representing control flow and data flow. Figure 3.3 exemplifies the different nodes and edges using the PDG for the Ansible playbook in Listing 3.1. It also highlights that the PDG representation enables whole-program analyses, as the graph contains nodes and edges related to the role included by the playbook.

Control Nodes and Control-Flow Edges

Action control nodes represent the action executed by each task. These are exemplified as two ellipses in Figure 3.3, one for each task. *Order* (`ORDER`) control-flow edges connect these control nodes, representing possible execution paths. An action node can have multiple outgoing order edges, i.e., multiple control-flow successors, which represents a fork in control flow caused by a conditional branching point. Similarly, an action node can have multiple control-flow predecessors, representing a join in previously-branched control flow.

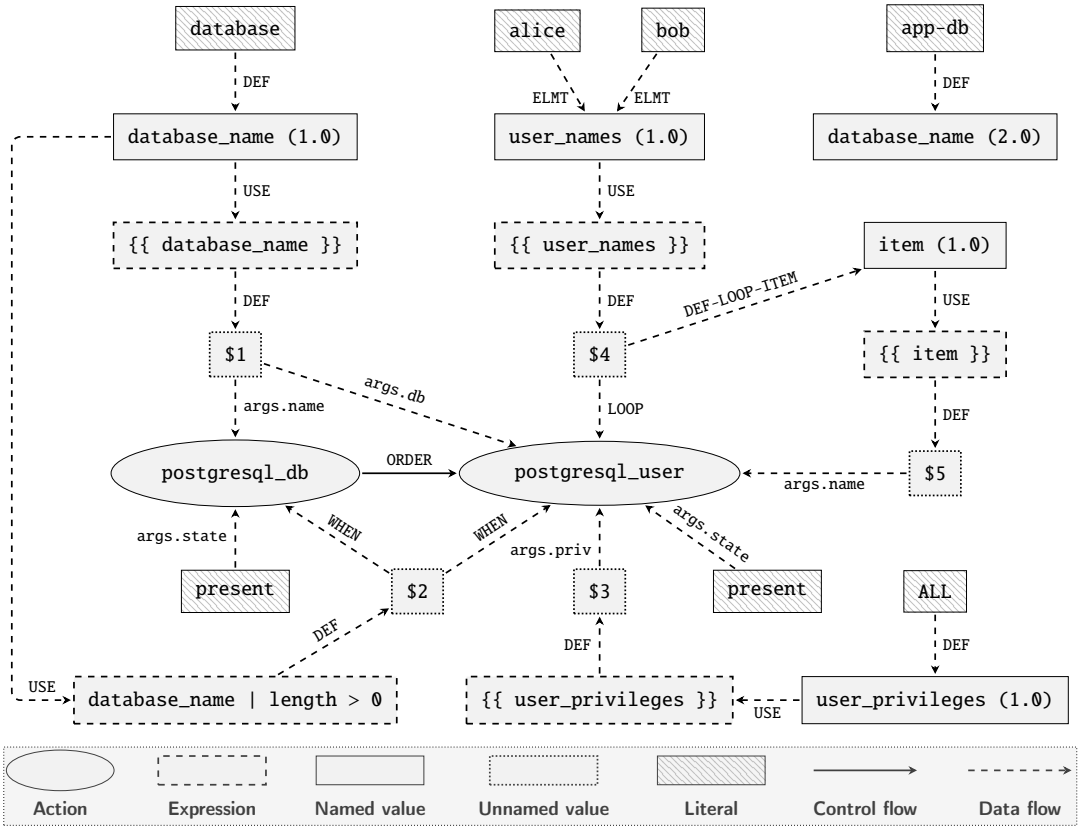

FIGURE 3.3: Program Dependence Graph of the Ansible playbook in Listing 3.1.

Data Nodes

Template expressions are represented as *expression* nodes. Furthermore, we distinguish between three types of data value nodes in the PDG. *Literal value* nodes correspond to concrete literals in the Ansible script. *Unnamed value* nodes represent abstract intermediate values produced by an expression. These can either be used directly by a control node, or they may be bound to a variable in a definition. *Named value* nodes represent abstract values produced whenever a variable is looked up and its initialiser expression is evaluated, and carry the name of the variable.

Importantly, the named and unnamed value nodes represent abstract values rather than concrete ones. However, the concrete value associated with one of these value nodes should remain the same during the execution of a script. Nonetheless, the same expression may produce different values at run time, and variables may be redefined. Such different values must not be conflated in a single named or unnamed value node. While unnamed value nodes always have unique identifiers, the PDG may contain multiple named value nodes with the same name. Therefore, we embed two additional identifiers in named value nodes, which together allow every named value node to represent a single abstract value.

The first is the *lexical definition version*, which changes when a new variable definition with the same name is introduced, enabling us to distinguish

between homonyms. Second, even for the same lexical definition, the value produced by dereferencing a variable anew may change due to the expression evaluation semantics (cf. Section 2.2.4). This can happen because the variable's initialiser is impure (e.g., an expression producing the current time), or because it uses other variables whose values have themselves changed. To distinguish between potentially different values for the same definition, we also embed a *value version* in the named value nodes.

Figure 3.3 depicts numerous data nodes of all aforementioned types. In the named value nodes, we denote the two additional identifiers as `d.v`, with `d` representing the lexical definition version and `v` the value version. For instance, the PDG contains two named value nodes for the `database_name` variable, with `1.0` defined in the playbook (Listing 3.1c), and `2.0` originating from the role's defaults (Listing 3.1b). Note that due to precedence, the `2.0` variable is never used, yet is still part of the program, and thus represented in the PDG.

Data-Flow Edges

Data nodes are connected through data-flow edges whose labels indicate the type of data flow that occurs. These can be categorised into three main types, namely data definition, data usage, and control/data dependence edges.

Definition (`DEF`) edges represent definitions of values, such as from an expression to an unnamed value, or from an unnamed value to a named value. *Looped definition* (`DEF-LOOP-ITEM`) edges are special-case definition edges that connect a value that is iterated over to the named value node that represents the loop variable, often named `item`. *Composition/element-of* (`ELMT`) edges connect two data nodes, in which the value represented by the origin node is an element of the value represented by the target. These are used to link lists and dictionaries to their items.

Conversely, *use* (`USE`) edges represent usages of values in an expression. *Argument* edges (denoted `args.`*`argument-name`*) are special-case use edges that connect an unnamed value or literal value to action nodes, and represent action arguments. Note that named values cannot be at the origin of an argument edge, as variables always need to be dereferenced by an expression first.

Finally, control/data dependence edges relate data flow to control flow, and vice versa. *Loop* (`LOOP`) edges connect a data node to an action that is executed iteratively for each item in the value represented by the data node. *Conditional* (`WHEN`) edges connect a data node to a node that depends on the value of the data node, and represent `when` directives. The target node may be an action, indicating conditional execution of a task. It may also be another data node, indicating that the target data node is defined conditionally, which may occur when a `set_fact` or `include_vars` task is executed conditionally.

The example in Figure 3.3 illustrates these data-flow edges. For instance, we can see that the named value node labelled `database_name (1.0)` is defined

using a literal. Moreover, two outgoing use edges link it to two expressions. The first simply dereferences the variable, defining the unnamed value node labelled `$1`, which is subsequently linked to the two action nodes using argument edges. The second expression represents the conditional, and produces an unnamed value node labelled `$2`. This latter node is then linked to the two actions using conditional edges. Note that Ansible in both cases would evaluate both expressions anew for the second task, but both would produce the same value, and are therefore represented only once in the PDG. This succinctly represents that the two actions are both data-dependent on the same data (`$1`) through their arguments, while also being control-dependent on the same data (`$2`) through the conditionals.

A second interesting situation is depicted around the `user_names` variable, showing composition and looped definition edges, as well as a loop control dependence edge. The loop edge represents that the second action is executed iteratively for each item in the `user_names` variable. Moreover, through the looped definition edge, we can see how the action uses the elements over which it iterates.

Comparison to PDGs in General-Purpose Programming Languages

The design of our PDG for Ansible code diverges from the original definition [38] of PDGs, both to account for Ansible's idiosyncrasies and to enable advanced software quality assurance. Originally, PDGs were described as graphs whose nodes represent program statements, and whose edges represent control or data dependence between two program statements [38]. In fact, the original PDG description does not explicitly represent data values as nodes. Later, PDGs were built upon in new representations, such as the Graph-based Object Usage Model (Groum) [80] and the Abstract Control-Data Flow Graph (ACDFG) [3]. Both of these representations added several new node and edge types, including data nodes and data-flow definition and usage edges. This motivated the addition of explicit data nodes and data-flow edges in our PDG design.

Moreover, the original PDG definition includes control-dependence edges, i.e., edges between two statements where the second statement is conditionally executed based on the result of the first statement. In our PDG representation, such control-dependence edges are replaced by control-flow order edges, as Ansible does not contain control-flow constructs akin to those typically found in general-purpose languages (e.g., if-statements and for-loops). Furthermore, the original PDG definition considers expressions as standard statements and thus represents them as standard control-flow nodes, whereas our PDGs instead encode expressions as data-flow nodes.

3.3.2 Building a Program Dependence Graph

Our PDG builder takes as input an Ansible playbook or role. It first parses it into a structural model representation, either in strict or lenient mode, as described in Section 3.2. Then, while traversing the structural model in a

depth-first manner, it adds nodes and edges to the graph to represent the control-flow and data-flow information. Throughout this, it takes into account Ansible's data-flow semantics, while also handling the control-flow jumps caused by inclusions of other files.

Representing Control-Flow Information

While traversing an Ansible project, the builder maintains *context* through a tuple comprising three individual context elements.

- The *data-flow context* is used to record variable definitions, to manage environments, and to abstractly evaluate expressions to produce data nodes. It is described in more detail later, when we discuss how data-flow information is represented.
- The *inclusion context* records information about file inclusion, e.g., through meta-actions such as `include_tasks`. It is used to look up and parse files in the project when one is included, and to raise errors when a circular dependency is encountered.
- The *control-flow context* keeps track of control flow during traversal of the tasks. It contains a set of control nodes that form the predecessors for the next task processed by the builder. The builder uses these to add control-flow order edges when processing tasks, and updates this set for the next task. Moreover, while descending in the depth-first traversal, the control-flow context stores the data nodes representing the results of conditional and loop expressions. The builder consults these to add loop and conditional control/data dependence edges in every task it processes.

The builder invokes specific routines depending on the type of element encountered. We describe each of these below. Algorithm 1 illustrates their working and the interactions with the different contexts using a pseudocode description of the first of these routines, namely the processing of generic tasks.

Generic tasks To process a generic task, the builder first uses the data-flow context to create the task's environment and to define the task's variables in this environment. If the task contains a loop directive, it uses the data-flow context to abstractly evaluate the loop expression. It stores the resulting data node in the control-flow context to indicate the loop control/data dependence, and uses the data-flow context again to define the iteration variable. If the task contains a conditional directive, the builder equally abstractly evaluates the conditional expression and stores the result in the control-flow context analogously. Subsequently, it adds an action node to the graph, and links the control-flow edges by consulting the control-flow context. Specifically, it uses the set of predecessors to add order edges, and the conditional and loop data nodes to add control/data dependence edges. Finally, it updates the control-flow context's set of predecessors with the newly-added action node.

Algorithm 1 Processing of generic tasks.

```
1:  procedure Process-Generic-Task(t, df, cf, g)
2:      input: task t, data-flow context df, control-flow context cf, PDG g
                                                          ▷ Enter new environment
3:      Activate-New-Environment(cf, df)
                                                ▷ Define task's variables in data-flow context
4:      for (name, init) ∈ t.vars do Define-Variable(df, name, init)
                                                                ▷ Activate loop, if any
5:      if t.loop ≠ ∅ then
6:          n_loop ← Resolve-Expression(df, t.loop)
7:          Activate-Loop(cf, n_loop)
8:          Define-Variable(df, "item", n_loop)
                                                            ▷ Activate condition, if any
9:      if t.when ≠ ∅ then
10:         n_when ← Resolve-Expression(df, t.when)
11:         Activate-Condition(cf, n_when)
                                                              ▷ Create task's action node
12:     n_action ← Create-Action-Node(g, t.action)
                                        ▷ Link control-flow and control/data dependence edges
13:     for n_predecessor ∈ cf.predecessors do Add-Order-Edge(g, n_predecessor, n_action)
14:     for n_condition ∈ cf.conditions do Add-Conditional-Edge(g, n_condition, n_action)
15:     for n_loop ∈ cf.loops do Add-Loop-Edge(g, n_loop, n_action)
                    ▷ Update predecessors, keeping previous predecessors if task is conditional
16:     if t.when ≠ ∅ then Set-Predecessors(cf, n_action)
17:     else Add-Predecessors(cf, n_action)
                                                                ▷ Process action arguments
18:     for (arg, value) ∈ t.args do
19:         n_arg ← Resolve-Expression(df, value)
20:         Add-Argument-Edge(g, n_arg, n_action, arg)
                                                            ▷ Process registered variable
21:     if t.register ≠ ∅ then Define-Fact(df, t.register, n_action)
                                ▷ Deactivate conditions, loops, and variables added for this task
22:     Deactivate-Environment(cf, df)
```

If the task is executed conditionally, the previous predecessors will also be kept, to indicate that the newly-added action node may be skipped.

Having processed the control-flow information, the builder turns to represent the task's data flow. To this end, it uses the data-flow context to abstractly evaluate the other expressions in the task's directives, such as action arguments. It links the obtained data nodes using argument edges. If the task has a `register` directive, which binds a variable to the result of its execution (cf. Section 2.2.4), the builder defines it in the data-flow context, which links it to the task with a data definition edge. Finally, to leave the task's execution environment, the builder deactivates the task's environment it added to the data-flow context and removes any looping and conditional control/data dependences it added to the control-flow context.

Tasks with meta-actions Meta-actions are handled as special cases, as they have an impact on the control flow or data flow of Ansible code. Like for generic tasks, the builder first defines the task's variables in a local environment with the correct precedence[2], and extracts conditional and loop control/data dependences. However, contrary to generic tasks, the builder does not add an action node for the task. Instead, further processing depends on the specific meta-action.

For task inclusion meta-actions, including `include_tasks` and `import_tasks`, the builder uses the inclusion context to find and parse the included task file. If the file cannot be found, e.g., because it is missing or because the file name is in fact an expression, which cannot be statically approximated, the builder ignores the meta-action. Otherwise, it proceeds with processing the task list by delegating to the appropriate routines for tasks or blocks. By having registered the control/data dependences in the control-flow context, any conditions or loops applied to the meta-action will automatically be applied to the included tasks by the other routines. Note that included tasks may themselves use inclusion meta-actions that are processed identically, thus establishing a depth-first traversal of the included file tree[3].

The role inclusion meta-action (`include_role`) is processed similarly to task inclusion. However, the inclusion context not only searches for the role in the Ansible project itself, but also in a user-configurable search path. This enables the PDG builder to also consider third-party Ansible roles, further establishing a whole-program analysis.

The variable inclusion meta-action (`include_vars`) is also processed similarly. However, rather than adding new action nodes, the builder uses the data-flow context to define the variables specified in the file, similarly to how it would define task variables.

Finally, the `set_fact` meta-action, which eagerly defines variables, is processed by first abstractly evaluating the variables' initialiser expressions, similarly to action arguments for generic tasks. The builder then defines the variables in the data-flow context, and links the data nodes obtained by evaluating the initialisers to the named variable nodes obtained by defining the variables.

Blocks To represent blocks, the builder first uses the data-flow context to define the block's environment and its variables. Then, it processes each contained task or nested block by delegating to the appropriate routine. If the block contains exception handling mechanisms, the PDG builder also processes the tasks for these mechanisms, adjusting the control-flow context's set of predecessors appropriately.

Playbooks and Plays For playbooks, the builder traverses each play in a depth-first manner and creates a disconnected subgraph of the PDG to

[2] The precedence depends on the type of meta-action. The environment for inclusion meta-actions have "include parameter" precedence, others have standard "task variable" precedence.

[3] Technically, file inclusion can be recursive. However, all recursive inclusions we have encountered in practice constituted defects. Therefore, the PDG builder considers a circular inclusion an error.

represent the play. It uses the data-flow context to define the play variables. Then, it processes each task, role, and handler in the execution order specified by the play, delegating to the appropriate routine and using the control-flow context's set of predecessors to link them together.

Roles To process roles, the builder first uses the data-flow context to define the variables in the role's defaults and role variables files. Subsequently, it traverses the role's main tasks file and handlers file, delegating to the appropriate routines for both.

Representing Data-Flow Information

As mentioned above, to derive data-flow information, the PDG builder maintains a data-flow context during its analysis. The data-flow context contains a collection of environments that store *data-flow records* which enable the builder to compute data dependences. This collection starts with an empty environment for each global environment (e.g., non-persistent facts, host variables, etc., cf. Section 2.2.4). The environment collection is expanded with new environments for every local environment entered, such as those for block or task variables, which are subsequently removed when the builder leaves the environment. When looking up a variable in the environment collection, the environments are traversed from higher to lower precedence until a matching definition is found.

Data dependences are computed from three types of records:

- Variable definition (VDef) records represent variable definitions with their initialisers. These are stored within the aforementioned environment collection.
- Expression value (EVal) records uniquely represent abstract values produced by an expression. To achieve this, they contain the expression's data dependences and a version number to distinguish between values produced by impure expressions. Effectively, these represent unnamed value nodes in the graph, and any change in an EVal record constitutes a new unnamed value node.
- Variable value (VVal) records uniquely represent abstract variable values and combine a VDef and an EVal record. These map to the named value nodes in the graph. In particular, the VDef part maps to the *lexical definition version* contained in the named value node, while the EVal record maps to the *value version*.

Two operations govern data dependence computation, namely *defining variables* and *abstractly evaluating expressions*. The former takes a variable name and its initialiser and inserts a variable definition (VDef) record representing the definition into the appropriate environment, depending on the definition's precedence. For variables defined in global environments, it also consults the control-flow context and, if any conditions are active, adds condition edges to signify that the variable is conditionally defined. Note that, since

Algorithm 2 Abstract expression evaluation.

```
 1: function Resolve-Expression(e, env)
 2:     input: expression e, environment collection env
 3:     output: EVal record r_e
 4:     d ← ∅                                          ▷ Set of data dependences

 5:     for n ∈ Get-Variable-References(e) do          ▷ Resolve all variables
 6:         d ← d ∪ Resolve-Variable(n, env)

 7:     if Is-Pure-Expression(e) then
 8:         r_e ← Create-EVal-Record(e, d, 0)
 9:     else                                           ▷ Different value caused by impure expression
10:         v ← Get-Next-Value-Version(e, d)
11:         r_e ← Create-EVal-Record(e, d, v)

12:     return r_e

13: function Resolve-Variable(n, env)
14:     input: variable name n, environment collection env
15:     output: VVal record r_v
16:     r_d ← Find-VDef-Record(env, n)

17:     if Initialiser-Is-Expression(r_d) then         ▷ Evaluate initialiser
18:         e ← Get-Expression(r_d)
19:         r_e ← Resolve-Expression(e, env)
20:         r_v ← Create-VVal-Record(r_d, r_e)
21:     else                                           ▷ Constant initialiser, e.g., set_fact
22:         r_v ← Create-Constant-VVal-Record(r_d)

23:     return r_v
```

initialiser expressions are evaluated lazily, the PDG builder does not evaluate the initialiser at definition time.

The latter operation takes an expression, abstractly evaluates it, produces and stores the appropriate records, and returns an unnamed value node representing the result. It also expands the PDG to represent the intermediate data nodes and data-flow edges resulting from the abstract evaluation. Note that the algorithm does not evaluate an expression to its concrete run-time value, but statically analyses the records in the environment collection to compute a unique abstract data value representing the expression's possible run-time values. Algorithm 2 describes abstract expression evaluation in pseudocode.

At a high level, Algorithm 2 uses two mutually-recursive functions, Resolve-Expression and Resolve-Variable. The former function resolves all variable values referenced by an expression using the latter function (lines 5–6), and produces an EVal record. Conversely, the latter function produces a VVal record by looking up a VDef record (line 16) and resolving its initialiser through

Resolve-Expression. If the variable definition has no initialiser to be evaluated (e.g., variables defined through `set_fact` or the `register` directive), it returns a constant VVal record instead (line 22).

The records returned by these functions uniquely identify abstract values used in the Ansible script. If two subsequent applications of Resolve-Expression produce identical EVal records, it is guaranteed that the concrete values during execution will be identical. This enables the PDG builder to represent fine-grained data dependences, since each EVal and VVal record uniquely maps to an unnamed or named value node in the PDG respectively.

However, care must be taken with impure expressions, as their value may change arbitrarily. The algorithm uses the value version stored in the EVal record to distinguish the different values produced by an impure expression (lines 10–11). In such cases, the PDG builder will reuse only the expression node, since data dependences remained the same, but will create new unnamed value nodes to represent the changed values produced by the expression.

To determine expression purity, our algorithm consults a set of built-in filters, tests, and "lookup" call names which are known to be pure. We distilled this set from the Ansible documentation, its source code, and experience. For example, the `first` filter, which returns the first element of a sequence, is contained within this set. However, filters such as `random` or tests such as `exists` (which checks whether a given path exists on the file system) are not in this set, as they are not pure because of their reliance on internal or external state. We consider a template expression to be pure if each filter, test, and "lookup" call it uses is within this set, which necessarily under-approximates the pure expressions in an Ansible program. Therefore, this straightforward approach can never mark an expression as pure while it is not, but can suffer from false negatives when user-defined filters or tests are used within a template expression.

In summary, using the operations to define variables and abstractly evaluate expressions, combined with mapping the data-flow records to data-flow nodes in the PDG, the PDG builder can accurately represent Ansible's intricate data-flow semantics. Together with the control-flow information accounted for, including inclusion of other files and even third-party code, the obtained Program Dependence Graphs constitute the most semantically-rich and most accurate code representation for Ansible to date.

3.3.3 Technical Limitations

Certain operations are too dynamic for the PDG builder to statically approximate. This includes dynamically including tasks (`include_tasks`) or variables (`include_vars`) where the file name of the included file is not a literal. The builder currently ignores such actions, and any tasks or variables that are included in this manner may not be represented. Similarly, although we indicate when variables are conditionally defined, the PDG builder does not consider the conditions under which variables may be defined when resolving

variable references. It may therefore use the wrong variable definitions under certain circumstances. We leave properly resolving conditional definitions as future work.

Finally, the heuristics used to determine whether an expression is pure are naive and can lead to an under-approximation. For instance, it does not support user-defined filters and tests, and considers them impure by default. We do not consider this to be an inherent limitation of our approach, since the implementation of this algorithm can be interchanged with an improved implementation without substantial changes to the PDG builder. However, an improved implementation aiming to automatically determine purity of user-defined filters and tests would need to perform complicated analysis of non-Ansible code. Alternatively, if the current algorithm was to be used in a practical tool implementation, it would be straightforward to allow a user to configure their own list of pure tests and filters.

3.4 Conclusion

In this chapter, we first reviewed the state of the art in static representations for Infrastructure-as-Code artefacts. We found that existing representations can be categorised as either *syntactic*, resulting from parsing the code, or *semantic*, resulting from static analyses and capturing behavioural information. However, when applied to Ansible, these representations exhibit several limitations. Syntactic representations fail to take Ansible-specific syntax into account, resorting to merely parsing the YAML files. Similarly, semantic representations fail to take Ansible's semantics into account, resorting to approximations or heuristics. Moreover, they only consider individual files, while an Ansible project is usually an ensemble of interacting files. Finally, although existing semantic representations can represent control-flow or data-flow information, they do not represent them together, or their interactions. These limitations lead to inaccuracies and impracticalities, which are undesirable for practical static quality assurance approaches.

We alleviated these limitations by introducing two new code representations for Ansible. The first, which we named the *structural model* for Ansible, addresses the limitations of syntactic representations by including Ansible-specific information in the parsing of YAML files. We defined its structure and how it represents playbooks, roles, their variables, blocks, and tasks through composition of structural elements. Moreover, we described how Ansible's internals can be used to construct this structural representation, offering syntax normalisation and validation.

Our second representation, the *Program Dependence Graph* for Ansible, builds upon this structural model by encoding and intertwining control-flow and data-flow information in a graph. We enumerated the numerous control and data nodes, as well as the control-flow, data-flow, and control/data dependence edges connecting them. Then, we described a whole-program static analysis to build such PDGs. The analysis processes control-flow information to

represent the execution order of Ansible code, and to represent multiple files in a single graph, including files originating from third-party code. Moreover, by reasoning about the data flow of Ansible code, the analysis builds an accurate yet succinct representation describing how data values are defined and used. Finally, by combining the two, the PDG represents interactions between control and data flow, such as the data values that influence which control-flow branch is taken, and control flow influencing which data values are defined.

These representations lay the foundations for advanced static analysis of Ansible code. We shall therefore use them in the subsequent chapters, to detect code smells in Chapter 4 and security weaknesses in Chapter 5. Furthermore, the representations introduced in this chapter may prove useful beyond the applications presented in this dissertation. For instance, the structural model may be a convenient representation to carry out code transformations, such as refactorings or stylistic changes. Moreover, the lightweight yet accurate nature of the PDG enables its use in practical yet advanced tooling, such as plugins for code editors, defect detectors, and more.

Chapter 4

Detecting Ansible Code Smells

The correctness of infrastructure code is critical to the functioning of an application. Various outages can be traced back to defects in the infrastructure's configuration code. For instance, in 2020, Cloudflare, a major content delivery network (CDN) provider and hosting company, deployed a faulty network configuration change that caused severe congestion in their network infrastructure.[1] Consequently, many large online applications that used Cloudflare's services, such as news websites and social media platforms, became unreachable.[2] Similarly, in 2021, engineers at Facebook issued a faulty infrastructure command that took Facebook's entire network offline for several hours.[3] In fact, configuration defects constitute one of the major causes of cloud outages [41]. Moreover, defective configuration data is estimated to be the most common type of defect in infrastructure code [102]. Unfortunately, the lack of test and verification tools for infrastructure code hampers ensuring its correctness [40].

As described in Chapter 2, Ansible's variable and expression semantics is unlike that of many other languages. We posit that this semantics may cause data-related defects and maintainability issues that are unique to Ansible. For instance, anecdotal evidence suggests that the precedence of variable declarations in Ansible causes widespread confusion among practitioners.[4,5] Similarly, Ansible's expression evaluation semantics has already caused defects in popular Ansible projects.[6]

Listing 4.1 illustrates the latter through a simplified playbook, which is adapted from a real-world defect in an open-source project. It first defines a variable, `app_version`, whose initialiser produces the current timestamp (line 3). This variable is then dereferenced twice, first on line 7 to create a directory, and on line 13 to download source files into the same directory. Recall from Chapter 2 that variable initialisers are evaluated lazily, and are re-evaluated every time the variable is dereferenced. However, `app_version`'s initialiser is impure,

[1] https://blog.cloudflare.com/cloudflare-outage-on-july-17-2020/
[2] https://techcrunch.com/2020/07/17/cloudflare-dns-goes-down-taking-a-large-piece-of-the-internet-with-it/
[3] https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/
[4] https://github.com/ansible/ansible/issues/75616
[5] https://github.com/ansible/proposals/issues/127
[6] https://github.com/servergrove/ansible-symfony2/commit/2285f4d8e9f4da7b33d08ade38102869a45f1455

```yaml
- hosts: all
  vars:
    app_version: "{{ lookup('pipe', 'date +%Y%m%d%H%M%S') }}"
  tasks:
    - name: Ensure version directory exists
      file:
        path: "/app/{{ app_version }}/"
        state: directory

    - name: Pull sources into directory
      git:
        repo: "https://github.com/my/repo"
        dest: "/app/{{ app_version }}/"
```

Listing 4.1: Playbook illustrating a real-world practical defect caused by Ansible's expression evaluation semantics.

producing a different timestamp every time. Combined, this causes the two references to resolve to different values, which causes the two tasks to use different directories, in turn causing the defect.

Figure 4.1 depicts the Program Dependence Graph (cf. Section 3.3) for the example playbook. The PDG shows that the initialiser expression defines two separate unnamed values (`$1` and `$3`) rather than one, indicating that the expression is impure and the values it produces may differ (cf. Section 3.3.2). Each of these nodes is then used to define a separate named value node, both originating from the same definition, as indicated by their lexical definition version, but having a different value version due to the different unnamed values. They are then used in separate tasks, showing that these tasks rely on the same expression, yet use different values, thus indicating the defect. This example shows that the PDG contains the necessary information to identify such error-prone code constructs.

Inspired by similar real-world defects, in this chapter, we propose a catalogue of 6 *variable smells*, which are code smells related to Ansible's variable precedence and expression evaluation semantics. Moreover, we present an automated approach to detect such smells, based on the PDG representation. Using a prototype implementation, we empirically investigate the prevalence and lifetime of variable smells in open-source Ansible projects.

The proposed smells can be used by Ansible practitioners to judge the quality of infrastructure code. Our empirical results show that certain smells are widespread, calling for tool support to detect and repair these flaws. They also point to likely misunderstandings about Ansible's semantics and signify their negative impact on infrastructure code quality, which may aid language designers in building safer IaC languages.

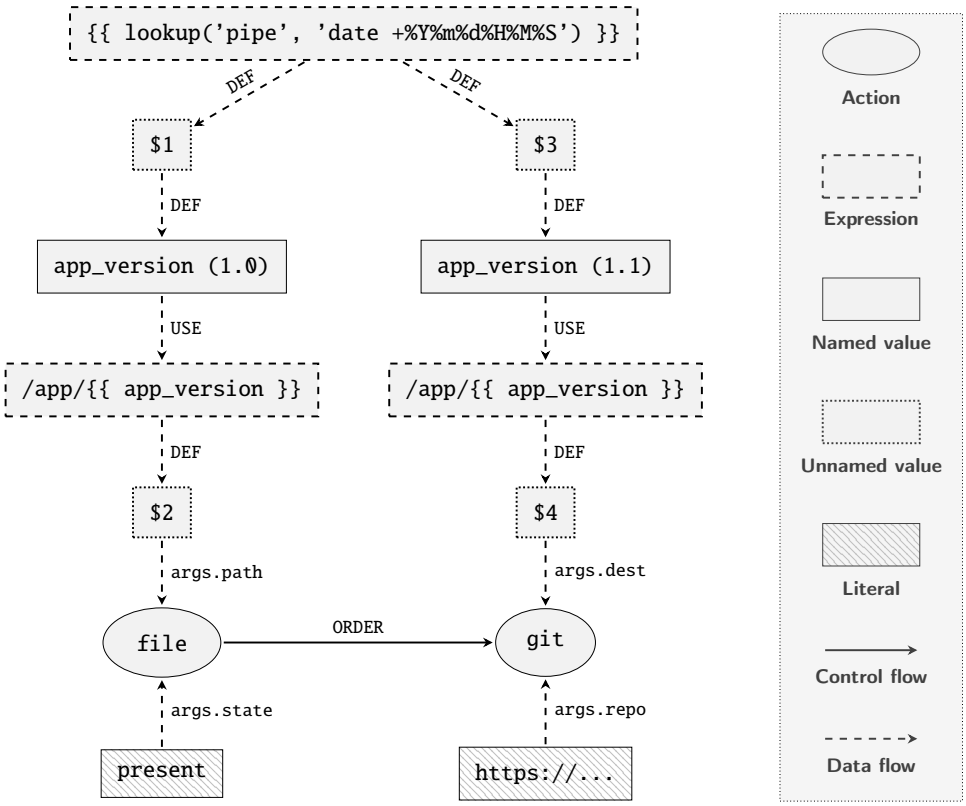

FIGURE 4.1: Program Dependence Graph for the playbook of Listing 4.1.

The remainder of this chapter is structured as follows. In Section 4.1, we review existing code smell detectors for Infrastructure as Code, and find that no existing detector takes the code's behaviour into account, making them unable to detect variable smells. Section 4.2 introduces the catalogue of variable smells we gleaned from Ansible's semantics. Subsequently, we describe our PDG-based approach to detect these smells in Section 4.3. We use this detector to perform a large-scale empirical study described in Section 4.4. Section 4.5 discusses the empirical study results. Finally, Section 4.6 concludes.

A replication package containing the data and analysis scripts used in this chapter is available at `https://doi.org/10.6084/m9.figshare.18819074`.

4.1 State of the Art

Several studies have focused on code smell detection for Infrastructure as Code. Sharma et al. [120] identify two types of smells for infrastructure code, namely *implementation smells* (e.g., complex expressions) and *design smells* (e.g., insufficient modularisation). They present a catalogue of such smells for Puppet and a tool that syntactically analyses Puppet code to identify them using an ad hoc implementation for each considered smell. Schwarz et al. [117] generalise Sharma et al.'s catalogue to other IaC languages, and similarly introduce a tool to detect design and implementation smells for Chef. Van der Bent et al. [135] perform a survey with Puppet developers to identify

aspects that influence the perceived quality of infrastructure code. Using the responses, they develop a quality model comprising syntactic source code metrics, and implement a tool to evaluate Puppet code against this quality model. Saavedra et al. [116] implement a polyglot code smell detection tool based on the GLITCH intermediate representation (cf. Section 3.1), which they extend to also represent Docker and Terraform. Dalla Palma et al. [27] focus on a specific code smell for TOSCA, namely *blob blueprints*, which are indicative of IaC files that are too large or too complex. They detect this smell using syntactic source code metrics as well as machine learning models. Finally, Kumara et al. [68] represent TOSCA deployment code as an architectural knowledge graph, and use graph querying to identify code smells taken from the aforementioned studies.

Code smells are not the only smells to have been studied. Hassan and Rahman [47] investigate test quality in IaC, and propose and detect 3 *test smells* that are indicative of problematic test cases. Several other studies have considered *security smells* [106, 108, 109, 113, 115] that indicate potential security weaknesses or vulnerabilities. These security smells are detected primarily through syntactic analyses with a so-called *rule engine* and a collection of logical predicates. Each predicate represents a rule to detect a certain smell. The rule engine traverses an abstract syntax tree of the code and applies each predicate on every visited node, marking smells when the predicate is satisfied.

All the aforementioned studies detect smells on a syntactic level. The only exception is the work by Kumara et al. [67], which uses an architectural model. Nonetheless, the considered smells are not caused by the semantics of the infrastructure code. This differs from the smells we are interested in studying in this chapter, which are directly caused by Ansible's unique semantics, and have not been studied before. Moreover, due to the smells being purely syntactic, the proposed detection approaches do not take an IaC script's behaviour into account. Therefore, detecting the behavioural code smells we are interested in requires a novel detection mechanism.

4.2 Catalogue of Variable-Related Smells

In this section, we describe 6 novel code smells, structured into 3 categories, concerning the usage and declaration of variables in Ansible code. Due to Ansible's unique variable precedence and expression evaluation semantics, their presence may cause confusion among IaC practitioners and lead to unexpected consequences and potential defects when the code is executed. Table 4.1 provides an overview of the smell catalogue, while Listing 4.2 depicts an example for each smell.

To construct this catalogue, we started from the assumption that the variable definition and expression evaluation semantics described in Section 2.2.4 may cause pitfalls for developers who are used to the semantics of general-purpose programming languages. Then, we used our knowledge of Ansible's semantics

```yaml
- hosts: all
  vars:
    my_var: "{{ 999 | random }}"
  tasks:
    - debug:
        msg: "{{ my_var }}"

    - debug:
        msg: "{{ my_var }}"
```

(A) Unsafe reuse: Impure initialiser (UR1).

```yaml
- hosts: all
  vars:
    var_a: "{{ var_b | upper }}"
    var_b: "abc"
  tasks:
  - debug: msg="{{ var_a }}"
  - debug: msg="{{ var_a }}"
    vars:
      var_b: "overridden!"
```

(B) Unsafe reuse: Changed data dependence (UR2).

```yaml
- hosts: all
  vars:
    my_var: "I'm a play var!"
  tasks:
  - set_fact:
      my_var: "I'm a fact!"
```

(C) Unintended override: Unconditional override (UO1).

```yaml
- set_fact:
    my_var: 123
- debug:
    msg: "{{ my_var }}"
  vars:
    my_var: 456
```

(D) Unintended override: Unused redefinition (UO2).

```yaml
- hosts: all
  tasks:
  - set_fact:
      my_fact: "{{ 1 + 1 }}"
```

(E) Too high precedence: Unnecessary `set_fact` (HP1).

```yaml
- hosts: all
  tasks:
  - include_vars: vars/other.yml
```

(F) Too high precedence: Unconditional `include_vars` (HP2).

LISTING 4.2: Contrived examples of each variable smell.

and personal experience to derive scenarios in which these semantics can lead to error-prone or defective code, establishing the first smell category. We subsequently extended the catalogue by devising new ways in which the established smells could be triggered, leading to the second and later the third category.

Unsafe reuse The first variable smell category, *unsafe reuse*, concerns the reuse of a variable whose value may have changed since a prior usage. These smells are derived from potential pitfalls caused by Ansible's unique expression evaluation semantics. When present, these smells may be indicative of a bug in case developers expected different occurrences of the same variable to evaluate to the same value, as is the case in the example of Listing 4.1.

Table 4.1: Overview of proposed code smells.

Category	Code	Smell name
Unsafe reuse	UR1	Impure initialiser
	UR2	Changed data dependence
Unintended override	UO1	Unconditional override
	UO2	Unused redefinition
Too high precedence	HP1	Unnecessary `set_fact`
	HP2	Unconditional `include_vars`

We discern two specific smells according to the root cause for the changed value. First, an unsafe reuse can be caused by an impure initialiser for the variable definition (UR1), as depicted in Listing 4.2a where the initialiser selects a random number (line 3). As such, the values used for the `msg` argument in the first task (line 6) will most likely be different from that in the second task (line 9).

Alternatively, an unsafe reuse can be caused by a change in the data dependences of the initialiser (UR2), which occurs when an upstream variable has been redefined. This is depicted in Listing 4.2b, where `var_a`'s initialiser refers to `var_b` (line 3). However, because `var_b` is overridden in the second task (line 9), the value to which `var_a`'s initialiser evaluates will be different in both evaluations, even though `var_a` itself is not redefined.

Unintended override The *Unintended override* category groups variable smells related to Ansible's variable precedence intricacies. We created these smells by investigating new ways to cause changes in data dependences that may lead to *unsafe reuse* smells. We discern two smells in this category, namely unconditional overrides (UO1) and unused redefinitions (UO2).

Unconditional override smells occur when a new definition overrides a previous definition at a higher precedence without taking the previous definition into account. It is inspired by "suspicious variable shadowing" smells for general-purpose languages. Listing 4.2c exemplifies this smell, where the definition of a fact through `set_fact` on line 6 unconditionally overrides an earlier play variable of the same name, defined on line 3.

The unused redefinition smell manifests itself as the definition of a variable that can never be used because a previous definition already exists at a higher precedence. This smell is derived from "unused variable" smells for general-purpose languages in combination with Ansible's complicated variable precedence system. It is illustrated in Listing 4.2d, where the task-local variable defined on line 6 can never be used, since the fact defined on line 2 will take precedence when the expression on line 4 is evaluated.

Too high precedence The final category, *too high precedence*, groups bad practices regarding the scoping and precedence of variable definitions. We created

these smells by investigating the mechanisms that Ansible provides to define high-precedence global variables and considering how these mechanisms can lead to *unintended override* smells, taking inspiration from "too broad scope" smells for general-purpose languages.

Recall from Section 2.2.4 that the `set_fact` and `include_vars` meta-actions allow variables to be defined dynamically, but that these variables have remarkably high precedences. Both of these meta-actions have valid use cases. The former can be used to eagerly evaluate impure expressions, whereas the latter can be used to dynamically decide whether variables need to be defined. However, executing these meta-actions will cause the variables to remain in a high-precedence global environment for the entire remainder of the playbook's execution. Therefore, we argue that these mechanisms of defining variables should be used sparingly.

Therefore, the *too high precedence* smells are intended to suggest possibilities to refactor such error-prone definitions. Specifically, the *unnecessary* `set_fact` (HP1) smell warns of variables defined through `set_fact` of which both the expression and all task conditions are strictly pure. Listing 4.2e depicts a contrived example, where the initialiser on line 4 performs a simple, pure calculation. Such definitions can be refactored to use lazily-evaluated initialisers rather than relying on high-precedence global definitions.

Similarly, the *unconditional* `include_vars` (HP2) smell indicates a variable that is defined using the `include_vars` meta-action without the use of any conditions. Line 3 of Listing 4.2f exemplifies such a usage. These unconditionally included variables can instead be defined through other means, like local variables, role variables, or play variables.

4.3 Detecting Variable Smells Using PDGs

We employ the PDG representation from the previous chapter (cf. Section 3.3) to detect the aforementioned variable smells. Our smell detector operates in two phases. First, it builds a PDG for each Ansible project given as input. Then, it traverses the PDG's nodes and applies a detection rule for each smell, akin to the rule engine design adopted by the state-of-the-art detectors for security smells described in Section 4.1. Table 4.2 depicts the detection rules for the smells.

Unsafe reuse smells are detected as pairs of named value nodes that originate from the same lexical definition (d_i) but have different value versions (v_i). The detection rules also compare the data dependences of the initialiser expressions that define the named values. When the data dependences are the same, the expression must be impure[7], and a UR1 smell is reported. Otherwise, the change in value is caused by a changed data dependence, leading to a UR2 smell.

[7] Otherwise, by construction of the PDG, the named value nodes would be the same.

TABLE 4.2: Detection rules for the proposed code smells. In the presented rules, we assume n, n_1, and n_2 are named value nodes, and that n_1 and n_2 share the same name. Symbols d_i and v_i refer to the lexical definition version and value version of n_i, respectively.

Smell	Detection rule
UR1	$d_1 = d_2 \wedge v_1 < v_2 \wedge getDeps(getDefExpr(n_1)) = getDeps(getDefExpr(n_2))$
UR2	$d_1 = d_2 \wedge v_1 < v_2 \wedge getDeps(getDefExpr(n_1)) \neq getDeps(getDefExpr(n_2))$
UO1	$d_1 < d_2 \wedge getPrec(n_1) \leq getPrec(n_2) \wedge n_1 \notin getDeps(getDefExpr(n_2))$ $\wedge\, getDeps(getCondExprs(n_1)) \cap getDeps(getCondExprs(n_2)) = \varnothing$
UO2	$d_1 < d_2 \wedge getPrec(n_1) > getPrec(n_2)$
HP1	$isSetFact(n) \wedge isPure(getDefExpr(n)) \wedge isPure(getCondExprs(n))$
HP2	$isIncludeVars(n) \wedge getCondExprs(n) = \varnothing$

getDeps(e) returns the set of data dependences for expression node e.
getDefExpr(n) returns the defining expression for named value node n.
getPrec(n) returns the precedence of n.
getCondExprs(n) returns the expressions controlling the conditional definition of n.
isSetFact(n) checks if n is defined through the `set_fact` meta-action.
isIncludeVars(n) checks if n is defined through the `include_vars` meta-action.
isPure(e) checks if expression e is pure.

For *Unintended override* smells, the detection rules compare two named value nodes of the same name but originating from different lexical definitions, indicating name clashes. The two detection rules differ in whether the first definition has higher precedence than the second (i.e., the first overrides the second) or vice versa. If the first definition has higher precedence than the second, then the second definition is unused, leading to the UO2 smell. Conversely, the UO1 smell triggers when the second definition overrides the first. However, UO1's detection rule also checks whether the second definition is unconditional. It considers the override to be unconditional if the first definition is not used in the new initialiser, and when there are no common data dependences in the conditions under which the two definitions exist.

Finally, the *too high precedence* smell detection rules match for variables that are defined using the considered meta-actions. The detection rule for HP1 emits a warning for every variable defined by `set_fact` of which both the defining expression and all conditions are strictly pure. Similarly, the rule for HP2 triggers for every variable defined by `include_vars` that is not defined conditionally.

4.4 Variable Smells in Practice

We now present our empirical study into the real-world prevalence and lifetime of the 6 proposed variable smells. We investigate the following research questions.

- *RQ_1: How precise is our code smell detector?* Before applying the code smell detector on a large scale, we need to assess its accuracy. To this end, we manually review a sample of reported code smells to estimate the detector's precision.
- *RQ_2: How prevalent are the proposed code smells in Ansible roles?* We aim to uncover how frequently these code smells occur in Ansible projects. Such information can help us understand which smells are the most problematic and should be prioritised during code maintenance.
- *RQ_3: Do the proposed code smells co-occur in Ansible roles?* We investigate co-occurrence of smells to gain a better understanding of potential correlation or causality between different smell types. For example, we hypothesise that variables defined with too high precedence may lead to unintended overrides.
- *RQ_4: What is the lifetime of a code smell in an Ansible role?* Understanding the lifetime of a code smell further aids in estimating the impact of these smells. For instance, we expect smells that are more defect-prone to have a shorter lifespan. It also provides insights into the practitioners' perspective by identifying which smells are fixed more often.

4.4.1 Dataset Collection

Our study focuses on Ansible roles, as roles can be reused by many projects, and a smell in a role may thus impact numerous client projects. We use Ansible's Galaxy ecosystem (cf. Section 2.2.6) to construct our dataset, as Galaxy is the official ecosystem of Ansible. We expect that it indexes the majority of the publicly-accessible, reusable roles. Therefore, we scrape the ecosystem to create a representative dataset comprising all roles on Ansible Galaxy.

Scraping Ansible Galaxy

To build the dataset, we construct a tool called Voyager.[8] Voyager's data collection and extraction pipeline goes through a number of phases. We briefly summarise these phases below.

In the first stage of the pipeline, Voyager polls various endpoints of the Galaxy API to collect raw metadata, e.g., the role endpoint for role information and the repository endpoint for git repository information. However, we encountered occasional internal server errors returned by the API, which we found to be

[8] Voyager is available at `https://github.com/ROpdebee/Voyager`

unrecoverable. To alleviate the issue, we additionally poll the role search endpoint, which does not exhibit these errors, and later deduplicate the data returned by both endpoints based on the role ID.

Voyager subsequently converts and normalises these raw API responses. Most notably, this phase removes attributes that are redundant in the sense that they can be derived trivially from already-captured information, e.g., the git clone URL from the GitHub URL. Moreover, it cleans up certain values, such as `null` to 0 when a number is expected, and converts timestamps to RFC 3339 format.

In the next phase, Voyager attempts to clone all git repositories present in the information collected in the previous phase. Any repository that failed to clone, e.g., because of invalid URLs or private repositories, is ignored. For the remaining repositories, it extracts information on the repository's development history, such as commit messages and dates.

We performed this collection on January 20th, 2021, and gathered 26 834 roles, their metadata (e.g., author, description), and their GitHub repositories. This represents all publicly-accessible roles on Ansible Galaxy at the time. All of this data, and more, is available in our dataset, named "Andromeda" [89].

Curating the data

The data we collected needs to be curated to remove duplicate roles. Therefore, we apply additional selection criteria. First, we remove 231 "mono-repositories", which are repositories that contain multiple roles. Such repositories pose a challenge as they may cause duplicate empirical findings, which would negatively influence our findings or cause bias. Second, we remove 3 438 forked repositories, which may similarly cause duplication and bias. After applying these criteria, our final dataset comprises 21 931 repositories and corresponding roles for analysis.

Then, we run the PDG builder (cf. Section 3.3.2) for each commit of each repository. We configure the PDG builder to parse its underlying structural model leniently, i.e., attempting to parse as many tasks as possible. Moreover, we do not provide a role search path, thereby instructing it to not attempt to resolve third-party included roles. This is necessary as we do not know which version of the third-party role an old version of the role would use. Running the PDG builder as instructed produces a total of 629 073 PDGs for individual snapshots of roles.

We perform a preliminary exploration of this dataset by investigating the types of variables that are used in the role snapshots. The results of this analysis are depicted in Table 4.3. We find that 19 661 roles (89.6%) have defined variables at least once in their history, and 19 393 still feature variables in their latest commit. Role defaults are the most common, while block variables are rare.

Finally, we run the variable smell detector on the produced PDGs. The resulting smell instances form the dataset of our study.

TABLE 4.3: Summary statistics of the types of variables used in roles.

	# unique roles		# variables per snapshot		
Variable type	**#**	**%**	**Mean**	**Std. Dev.**	**Median**
Block variables	89	0.41	4.2	5.1	2
Include parameters	9 949	45.37	1.2	1.3	1
Included variables	294	1.34	5.3	11.9	3
Role defaults	17 229	78.56	13.5	26.2	6
Role variables	4 780	21.80	5.2	8.3	3
Non-persistent facts	8 638	38.39	4.1	7.9	2
Task variables	1 157	5.28	2.7	4.3	1
Total	19 661	89.65	15.8	29.5	8

TABLE 4.4: Precision of the variable smell detector for the different smells.

Category	**Smell**	**# TP**	**Precision**
Unsafe reuse	UR1	17	85%
	UR2	20	100%
Unintended override	UO1	16	80%
	UO2	17	85%
Too high precedence	HP1	20	100%
	HP2	20	100%
Total		110	91.67%

4.4.2 RQ_1: How precise is our code smell detector?

Research method To validate our PDG builder and smell detector, we randomly sample and manually validate 20 unique detected instances of each proposed variable smell (i.e., 120 in total). We consider a smell instance to be a false positive if the detection rule should not have matched, but do not take the role developer's intent into account to eliminate subjectivity from the validation. For example, we consider *unconditional override* (UO1) instances to be true positives even when it seems that the developer intentionally overrides the variable, since the detector is intended to produce objective warnings, not all of which may truly be bugs.

Results Table 4.4 summarises the results. The detector achieves good precision for most smell types. Moreover, all the encountered false positives stem from limitations of the PDG builder. Specifically, the 3 false positives for the *unsafe reuse due to impure initialiser* (UR1) smell are caused by over-approximations of the PDG builder due to unrecognised filters in expressions, and can be remedied easily in future work. Similarly, the 4 false positives of *unconditional override* (UO1) instances are caused by a builder limitation related

TABLE 4.5: Smell instances during the lifetime of Ansible roles. The number of unique smells per role only considers those roles in which at least one such smell occurs.

		# affected roles		# unique smells per role	
Category	**Smell**	**#**	**%**	**Mean**	**Median**
Unsafe reuse	UR1	37	0.9	1.8	1.0
	UR2	30	0.7	2.2	2.0
Unintended override	UO1	2 124	49.9	5.1	2.0
	UO2	14	0.3	2.0	1.5
Too high precedence	HP1	3 345	78.5	5.7	2.0
	HP2	184	4.3	6.4	3.0

to multi-level task conditionals. The 3 *unused redefinition* (UO2) false positives are caused by the builder not recognising certain dynamic task inclusion actions, which caused it to assign the wrong precedence to some variable definitions. Finally, we find no false positives for the *too high precedence* smells.

Answer to RQ_1: A manual validation of 120 random smell instances shows that the detector achieves a precision of 92%. This precision is sufficient for the rest of the study.

4.4.3 RQ_2: How prevalent are the proposed code smells in Ansible roles?

Research method We investigate the prevalence of variable-related code smells in practice. Since subsequent snapshots may contain the same smell instances, we deduplicate them. To this end, we consider two instances occurring across two snapshots of the same role to be the same instance if both are of the same smell type and are caused by the same variable definition. Then, for each smell type, we calculate how many roles have been affected by such smells in at least one commit. We also calculate descriptive statistics of the unique number of these smells per role. Finally, we study the evolution of the cumulative number of smells over time, including the number of smells introduced and fixed. To this end, we order the role snapshots according to their commit date, and compare the set of smells across two subsequent snapshots to identify additions and removals.

Results Overall, we found 31 334 unique smell instances, spread across 4 260 (19.4%) roles and 109 719 role snapshots. Of these, 21 934 (70.0%) are still present in the latest version of the roles. Table 4.5 summarises the results. Unnecessary usages of `set_fact` (HP1) and unconditional overrides (UO1) are by far the most prominent smells. Conversely, both types of unsafe variable reuses occur rarely. This may suggest that UR1 and UR2 are more likely to result

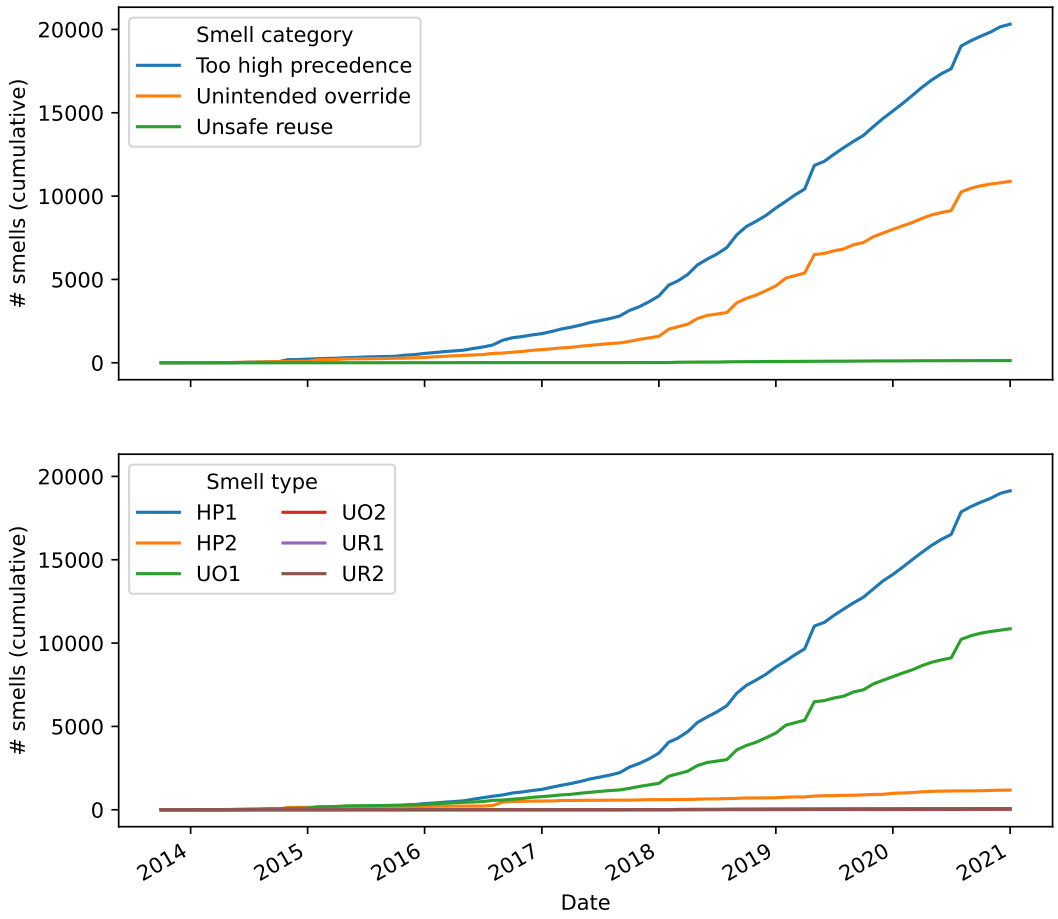

FIGURE 4.2: Cumulative number of smell instances over time, per category (top) and per smell type (bottom).

in defects, and are therefore fixed before they are committed to the repository. It may also be the case that most variables are only used once.

Figure 4.2 depicts the evolution of the smell prevalence over time. It is clear from the plot on the top that *too high precedence* and *unsafe override* smells continue to be introduced. The plot on the bottom shows that the HP1 and UO1 smells are the main reasons for the observed trends. Figure 4.3 compares the cumulative evolution in the number of added and number of fixed smells on a monthly basis. It shows that the rate at which new smells are introduced far outpaces that of their fixes.

Answer to RQ_2: 19.4% of the roles are affected by the proposed variable smells, and 70% of smells are still present in the role's latest snapshot. Most smells concern definitions with too high precedence, followed by unintended overrides. Unsafe reuses occur rarely. New smells are introduced more frequently than existing ones are fixed.

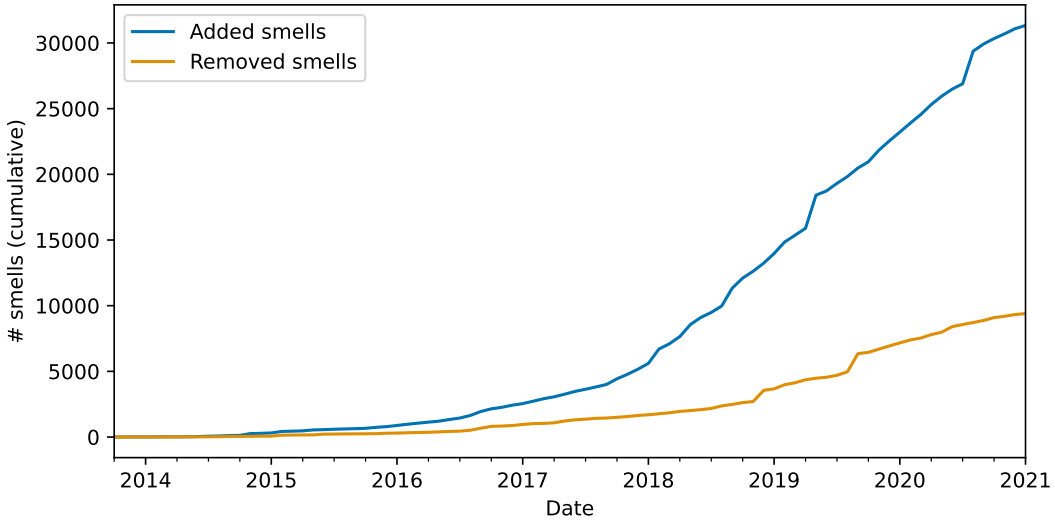

FIGURE 4.3: Cumulative number of new and fixed instances.

4.4.4 RQ$_3$: Do the proposed code smells co-occur in Ansible roles?

Research method We investigate co-occurrence between smells at two granularities. First, we consider co-occurrence at the role snapshot level, i.e., a single role snapshot that exhibits multiple types of smells. We calculate the proportion of roles that have at least one snapshot in which a co-occurrence exists. Second, to study whether smells co-occur within the same files in an affected snapshot, we calculate the proportion of files in a role version in which certain types of smells co-occur. When a file exhibits the same co-occurrences in different role snapshots, it is only considered in the first snapshot to avoid overestimating the file-level co-occurrences for roles with many commits.

Results Among the 109 719 role snapshots with smells, we found 43 255 (39.4%) in which at least two smells from different categories co-occurred, spread across 1 334 roles. Figure 4.4 shows the proportion of roles that had smells that co-occur within the same snapshot.[9] We observe *unintended override* smells co-occur with *too high precedence* smells more often than they occur alone within the same role snapshot. This finding suggests that variables defined with too high a precedence may lead to problems, including other smells. Nonetheless, as shown by the high proportion of roles in which smells occur alone, both can occur independently. Moreover, Figure 4.4 shows that many of the *unsafe reuse* smells co-occur with smells of at least one other type.

We also investigated how frequently smells co-occur in the same file. Figure 4.5 depicts the number of co-occurrences found in the files contained in role snapshots. We find that *unintended override* and *too high precedence* smell instances more often co-occur in the same file, than they occur in isolation. Other types of co-occurrences are rare.

[9] Note that the totals do not sum to 100% since it is possible for a role to have several snapshots that exhibit different co-occurrence types.

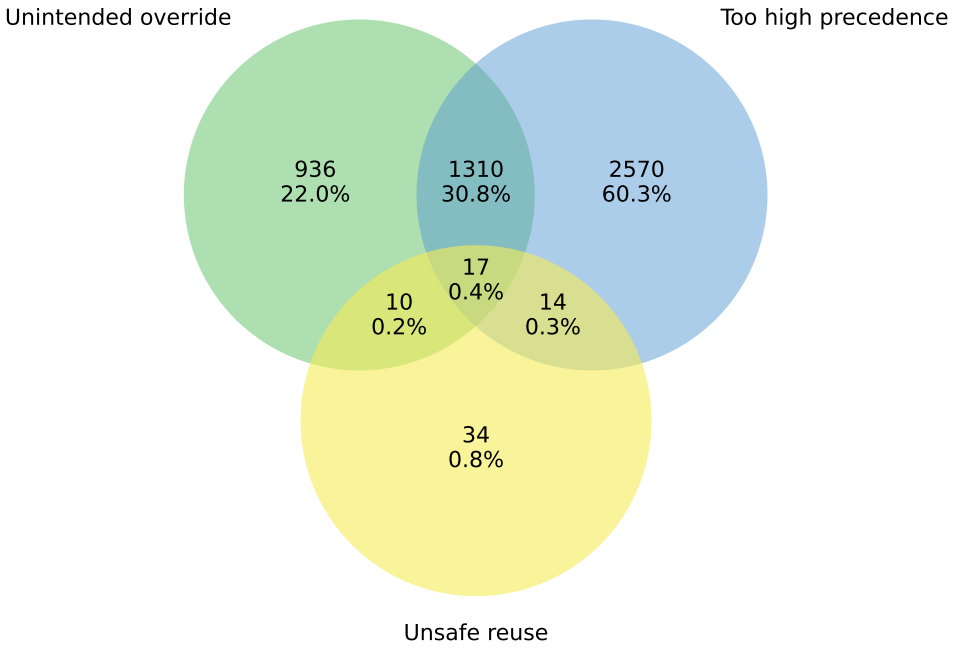

FIGURE 4.4: Venn diagram depicting the number and proportions of role snapshots that exhibit a type of smell co-occurrence, i.e., co-occurring across any of the files of a role at a certain commit.

Moreover, we investigated how many of a snapshot's files are impacted by co-occurring smells. To this end, Figure 4.6 depicts letter-value plots describing the proportion of files in a snapshot in which a co-occurrence can be found, for snapshots exhibiting such file-level co-occurrences. We find that co-occurrences of *too high precedence* and *unintended override* can frequently be found in most files in the snapshot. On average, such a co-occurrence impacts 79.2% files in a snapshot, whereas more than half of the co-occurrences impact all files of the snapshot (median 100%). However, note that many roles may have only a single file, thus trivially leading to 100% of their files being impacted. For instance, we only found 2 cases where all three smell types co-occur in a file, and in both cases, the role only contained a single file. Finally, when focusing on files without co-occurring smell categories, we find that the proportion of files impacted by *too high precedence* smells alone follows a uniform distribution. The other smell categories tend to impact proportionally fewer files in the snapshot. Specifically, *unintended override* smells impact an average of 45.5% and a median of 50% of files in a snapshot, whereas *unsafe reuse* smells impact an average of 32% and a median of 25% of files.

Answer to RQ_3: Although only 39.4% of roles exhibit co-occurring code smells, we find that *unsafe reuse* smells frequently co-occur with *too high precedence* smells. We also find that *too high precedence* and *unintended override* smells frequently co-occur within the same files.

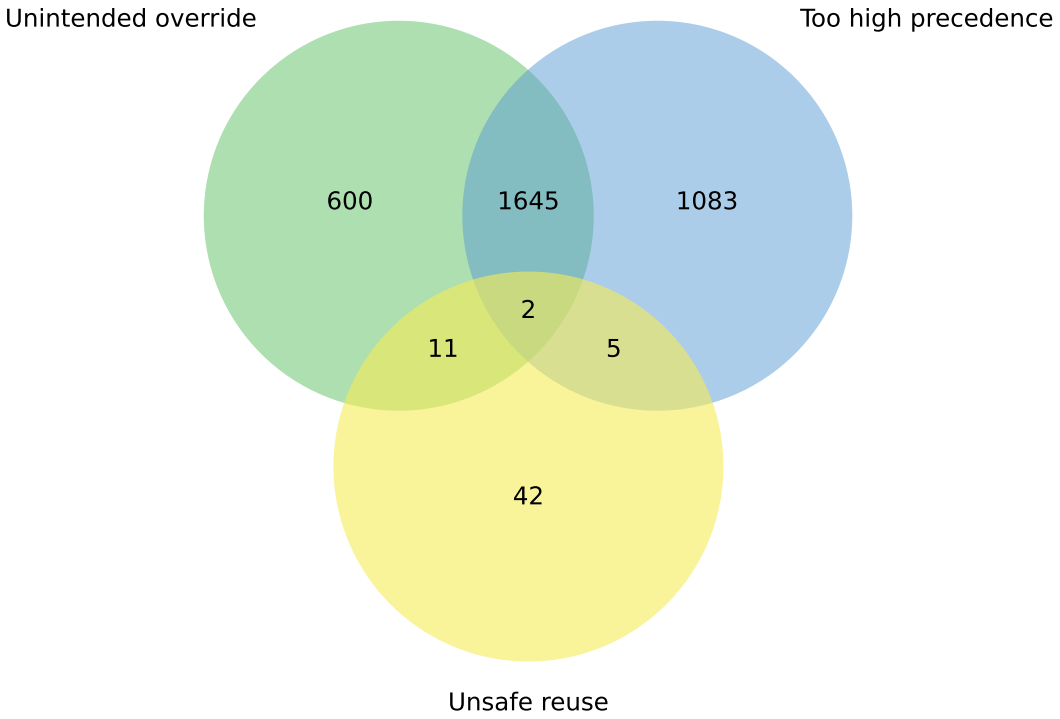

FIGURE 4.5: Venn diagram depicting the number of individual role files of a snapshot that exhibit a type of smell co-occurrence.

4.4.5 RQ_4: What is the lifetime of a code smell in an Ansible role?

Research method To understand the lifespan of variable smells, we investigated when they get introduced in a role and how long it takes for them to be fixed. First, we study how long it takes for a smell to be introduced into a role. To this end, we consider both absolute time (number of months) and relative time (proportion of commits) elapsed since the role's first creation, to account for different development speeds. Afterwards, we investigate how frequently smells are introduced in the role's first commit, and during the addition of new code files to the role.

To study fixes for variable smells, we investigate the amount of time required before a smell is removed since its original introduction. RQ_2 showed that the majority of smells are still present in a role's last snapshot and have thus not been fixed. Therefore, we use a survival analysis [64] to estimate the probability over time for a smell to be removed, with respect to the date of the first commit introducing the smell. Survival analysis creates a model that estimates the survival rate of a population over time until the occurrence of an event, in our case, the disappearance of the smell from the role. We use log-rank tests to confirm statistical differences between the survival rates of all considered categories of smells. We determine the p value using a Bonferroni correction. Specifically, starting from 95% confidence ($\alpha = 0.05$) and for $n = 18$

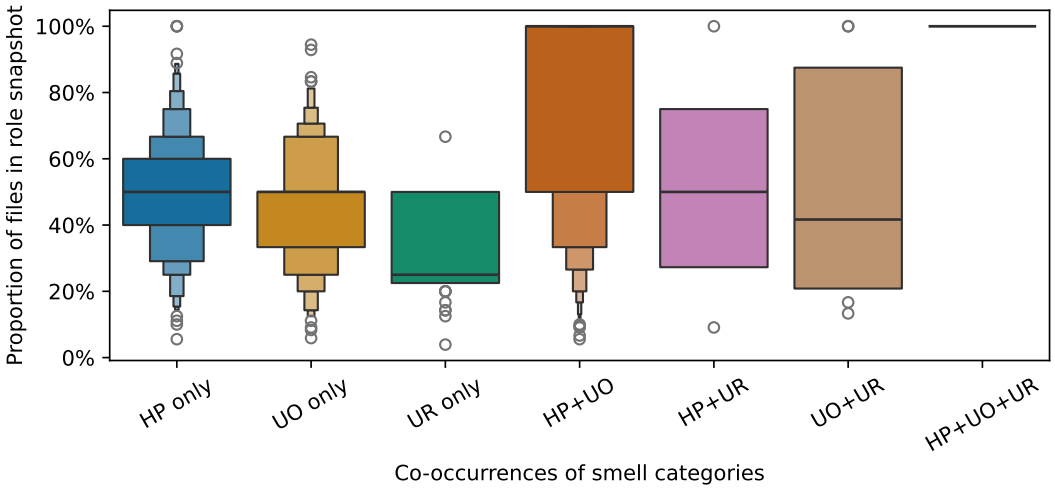

FIGURE 4.6: Letter-value plots depicting the distribution of the number of files in a role snapshot in which smell categories co-occurred. Note that each plot only considers role snapshots that exhibit a file-level co-occurrence, and that the distributions are depicted proportionally.

tests, we obtained a corrected p value of 0.002.

Results Figure 4.7 depicts the cumulative evolution of the first appearance of smells in function of the lifecycle of the affected role, i.e., the proportion of commits already made. We observe that smells are introduced at a steady pace throughout the role's entire development lifetime. Nonetheless, half of all smells were introduced in the first 30.7% of the roles' development lifecycles, because many smells are already present from the first commit to a role. Specifically, 16.8% of code smells are introduced in the first commit to a role. This affects 1 359 roles or 31.9% of all roles with smells. Of these roles, 82.3% contained *too high precedence* smells, 38.5% contained *unintended override* smells, while only 0.66% contained *unsafe reuse* smells.[10]

Figure 4.7 also shows that *too high precedence* smells are the first to appear, at a median of 27.8% of the role lifetime, opposed to 33.8% and 35% for *unintended override* and *unsafe reuse*, respectively. This may again suggest that *too high precedence* smells can cause the other smells. Moreover, as can be seen on the bottom figure, most HP2 smells are introduced early in the development cycle, with half being introduced in the first 18.1% of the roles' lifetimes. In absolute terms, we find that 50% of all smells are introduced within the first 0.46 months of the role's lifetime. Specifically, it took 0.3, 0.96, and 4.05 months for 50% of *too high precedence*, *unintended override*, and *unsafe reuse* smells to appear, respectively. This large discrepancy between the absolute and relative time periods for the *unsafe reuse* smell may indicate that it is more likely to occur in roles with longer development cycles, and were introduced when Ansible's semantics was less understood.

[10] Note that a project might have code smells of different categories.

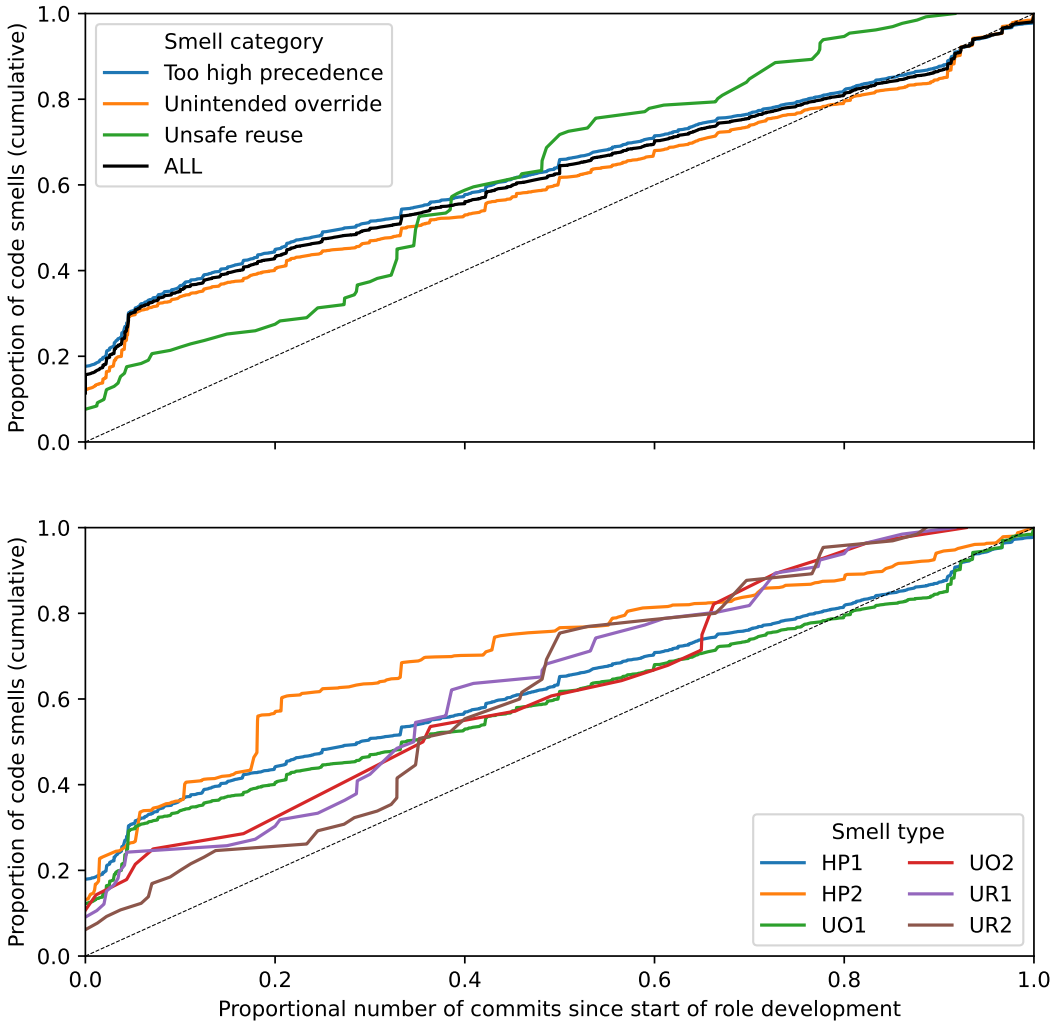

FIGURE 4.7: Proportion of code smells in function of the proportional number of commits made since a role's first commit, per category (top) and per smell type (bottom).

In terms of code files, we find that out of the 58 881 code files, only 7 171 (12.2%) contained any smells. Table 4.6 depicts the number and proportion of those affected files in which a smell of a certain type has been present since its first introduction, as well as the number of roles and smells these correspond to. We observe that 55.3% of smelly files already contained these smells when the file was added to the corresponding role. Furthermore, 42.7% of *too high precedence* smells are introduced together with their encompassing file. Similarly, for 35.5% roles that contain *unsafe reuse* smells, at least one of these smells was introduced together with the file.

Finally, we investigated the amount of time required before a smell is removed since its original introduction. Figure 4.8 shows the Kaplan-Meier survival curves for our smells. The curves for *unintended override* and *unsafe reuse* overlap, which suggests that there is no clear difference in terms of fixing time for smells in these two categories. In contrast, the *too high precedence* curve does not overlap with any other. It also takes longer before smells of this

TABLE 4.6: Statistics about code files whose first version contained at least one smell.

	Files		Roles		Smells	
Category	**#**	**%**	**#**	**%**	**#**	**%**
Too high precedence	3 138	56.8	2 112	61.0	8 667	42.7
Unintended override	1 568	48.2	1 126	52.8	3 333	30.6
Unsafe reuse	25	29.4	22	35.5	35	26.7
All	3 966	55.3	2 580	60.6	12 035	38.4

category are fixed. It takes 17.6 and 12.8 months for 50% of *unintended override* and *unsafe reuse* smells to be fixed, respectively, while it takes 37.4 months for 50% of *too high precedence* smells to be fixed. Log-rank tests confirmed statistical differences between the *too high precedence* category and the other two categories. This difference is mainly due to HP1 smells (in the right figure). Conversely, the tests did not confirm such a difference between the *unintended override* and *unsafe reuse* categories ($p = 0.016$). Finally, considering all smells categories, we found that it takes 26.7 months for half of all smells to be fixed.

Answer to RQ$_4$: Code smells are introduced steadily throughout the role's lifetime. Half of the discovered code smells appeared within the first 30% of a role's lifetime, with 16.8% already existing in the first commit. However, smells take much longer to be removed, still having a survival probability of over 50% after more than 2 years.

4.5 Discussion

In this section, we discuss the practical implications of our findings (Section 4.5.1). We also present potential threats to validity in Section 4.5.2.

4.5.1 Practical Implications

The variable smells proposed in this chapter can be used by Ansible practitioners to identify maintainability issues and find potential defects. Almost a fifth of the roles in our dataset is affected (cf. RQ$_2$), and not seldom from their first commit onwards (cf. RQ$_4$). Next to a call to arms for better tool support and safer IaC languages, this suggests that there are likely misunderstandings about Ansible's semantics among practitioners.

Practitioners' perception of smells Although we have not consulted with practitioners for this study, we found various fixing commits (cf. RQ$_4$) suggesting that practitioners agree with these smells. For instance, one developer fixed an unnecessary `set_fact` usage to address an issue where built-in facts were

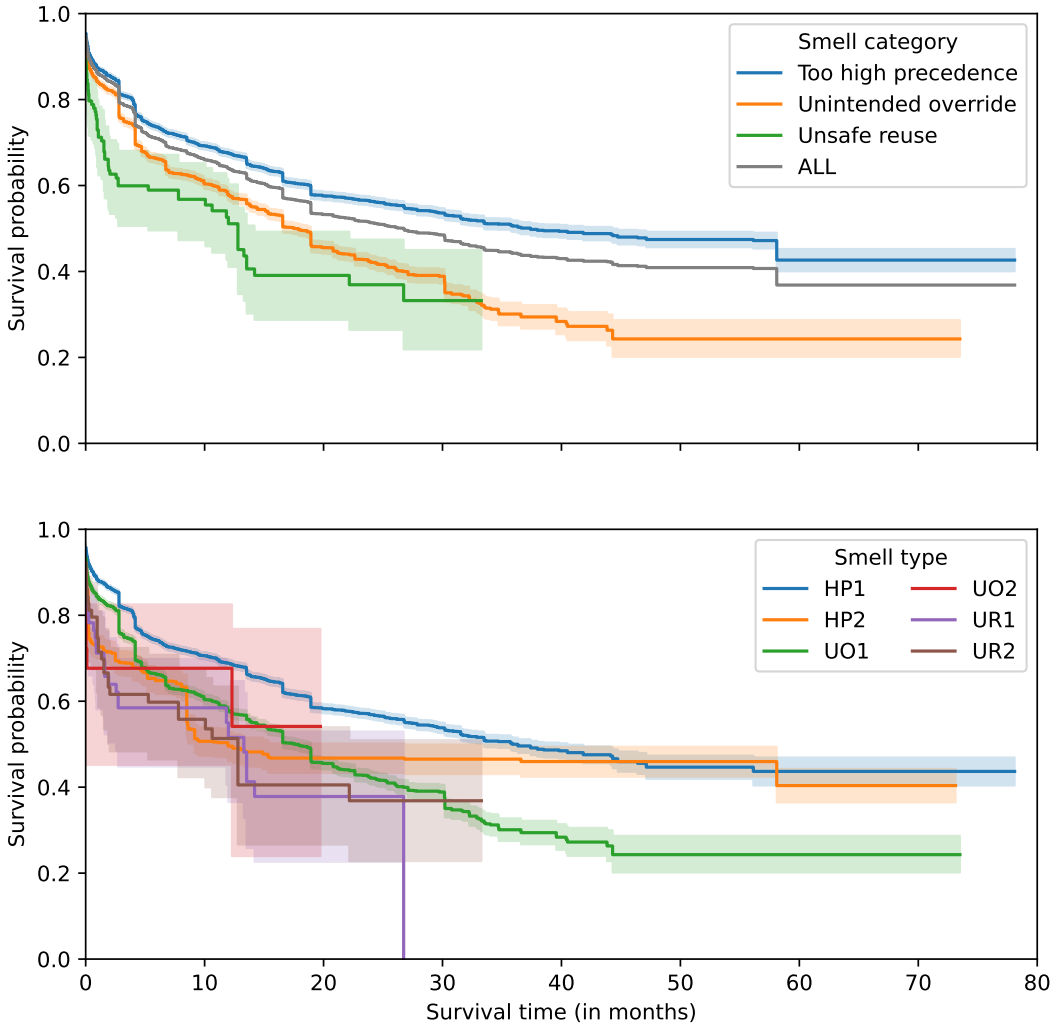

FIGURE 4.8: Kaplan-Meier curves showing the probability for a code smell to fixed in function of the time since the first commit in which the smell appeared, grouped by smell category (top) and name (bottom). The shaded coloured areas represent the confidence intervals ($\alpha = 0.05$) of the survival curves.

overridden.[11] Another developer removed three such usages to improve their role's quality score,[12] a metric computed by Ansible Galaxy to rate roles based on syntactic linter warnings. Similarly, various smells related to unsafe reuses of impure expressions correspond to defects that were later addressed.[13,14] This suggests that the discovered smells could be used to warn developers about potential defects and to recommend fixes. We leave this as future work.

[11] https://github.com/dj-wasabi/ansible-zabbix-agent/issues/207

[12] https://github.com/Senzing/ansible-role-stream-producer/commit/57f0f85f389aa93677465952d528040082d2cddc

[13] https://github.com/MonolithProjects/ansible-github_actions_runner/commit/94616c56a760f84e5738eac9b8c0534e935c2499

[14] https://github.com/servergrove/ansible-symfony2/commit/2285f4d8e9f4da7b33d08ade38102869a45f1455

```yaml
- include_tasks: types/pin.yml
  vars:
    # Include param
    apt__pin: ...
```

(A) `tasks/configure-preferences.yml`

```yaml
- template: ...
  vars:
    # UO2: Shadowed by include param
    apt__pin: ...
```

(B) `tasks/types/pin.yml`

LISTING 4.3: Example of a real-world UO2 smell in the *Turgon37.apt* role, commit `948c785c`.

Common pitfalls During the validation of the smell detector (RQ_1), we discovered certain patterns in the misunderstandings of Ansible's semantics. We discuss them as knowing which pitfalls to avoid can be valuable for practitioners. For instance, many of the *unnecessary* `include_vars` (HP2) instances were caused by practitioners explicitly loading files defining their default and role variables. Not only is this redundant since these files are loaded implicitly, it also unconditionally overrides the already-defined variables (leading to UO1 smells) and leads to unnecessarily high precedence, which may make it impossible for a client play to override these variable definitions to customise the role's behaviour. Moreover, many of the *unused redefinition* (UO2) smells were caused by the use of *include parameters*, which have the highest possible precedence. This is exemplified in Listing 4.3, where a task variable in an included file is shadowed by an include parameter defined on the `include_tasks` task.

Additional language features We also found apparent workarounds for a lack of language support for particular variable use cases, which can serve as the motivation to introduce new language features to address these use cases. A number of unnecessary `include_vars` (HP2) smells, although true positives, would be difficult to address since the dynamically loaded variable files were used to modularise variable definitions. This is exemplified in Listing 4.4, where the variables for different Amazon AWS services are divided into separate files that get included one after the other. Similarly, many unconditional overrides (UO1) affected variables defined as task results through the `register` directive. Although these variables were only used locally, `register` defines them globally, leading to conflicts as shown in Listing 4.5. Finally, all the sampled data dependence changes (UR2 smells) were caused by a form of dependency injection, where the initialiser of a global variable depended on a variable defined in the local environment of a task. Although many of these redefinitions seemed intentional, we classified them as true positives since the detector correctly identified the change in variable values and this usage of variables may be confusing. Nonetheless, this smell uncovered another instance of a real defect in an Ansible role, illustrated in Listing 4.6.

Additional tool support Our results can also serve to motivate future research on Infrastructure as Code maintainability. For instance, unnecessary

```yaml
# Contents of tasks/main.yml
- include_vars: infrastructure/subnets.yml
- include_vars: infrastructure/security-groups.yml
- include_vars: infrastructure/load-balancers.yml
- include_vars: infrastructure/instances.yml
```

Listing 4.4: Example of a real-world HP2 smell in the `telus.aws-infrastructure` role, commit `94c2203c`.

```yaml
# Contents of tasks/tasks_python_fallback.yml
- shell: yum -y install python3
  register: result # Define `result`
  until: result is not failed # Use `result` from line 3 locally

- shell: yum -y install python3-devel python3-setuptools
  register: result # UO1: Unconditional redefinition of `result`
  until: result is not failed # Use `result` from line 7 locally
```

Listing 4.5: Example of a real-world UO1 smell in the `softasap.sa_docker` role, commit `96548e70`.

usages of `set_fact` (HP1) are the most common smell we detected (cf. RQ_2). Due to its imperative nature and high precedence, we believe these to have a negative impact on the clarity and maintainability of infrastructure code. Moreover, our findings in RQ_3 suggest that *unintended override* smells may be caused such usages. Future work may investigate `set_fact`'s use cases, and possibly recommend refactorings to safer alternatives. Similarly, the results of RQ_2 show that smells are introduced more often than they get fixed, while RQ_4 shows that it may take a long time for a fix to arrive. Therefore, tool support for IaC practitioners to detect, comprehend, and repair these smells may be beneficial.

4.5.2 Threats to Validity

We present the threats to validity of the empirical study presented in this chapter following the classification and recommendations of Wohlin et al. [140].

The main threat to *construct validity* stems from the prototypes that we built to detect the smells. Our PDG builder suffers from technical limitations, discussed in Section 3.3.3. They may have introduced false negatives and false positives in our results. However, we have mitigated the threat of the latter by manually validating the precision of our prototypes on a random sample (cf. RQ_1), and we did not observe a large negative impact of its limitations during

```yaml
# Contents of tasks/configure-preferences.yml
- include_tasks: addkey.yml
  loop: "{{ keys }}" # Defines `item`
  vars:
    # These expressions should use `item` defined on line 3
    ssh_config_dir: "{{ item.ssh_config_dir }}"
    ssh_host_usewith: "{{ item.ssh_host_usewith }}"

# Contents of tasks/addkey.yml
- file:
    path: "{{ ssh_config_dir }}" # OK: Uses `item` defined on line 3
- blockinfile:
    # UR2: New value, expression now uses `item` redefined on line 15
    path: "{{ ssh_config_dir }}"
  loop: "{{ ssh_host_usewith }}" # Redefines `item`
```

Listing 4.6: Example of a real-world UR2 smell in the `dottgonzo.add_ssh_key` role, commit `7140f935`.

our validation. Nonetheless, we could not measure recall due to the absence of a ground-truth set of smells, meaning our findings may under-approximate the number of variable smells in practice. Second, the removal of a smell does not necessarily imply that it was fixed, as the containing code might have been deleted. The survival probabilities shown in RQ_4 thus form a lower bound.

Internal validity concerns choices and factors internal to the study that could influence the observed results. We decided to only study roles that are hosted on Ansible Galaxy, while there might be other roles available in GitHub but not distributed via Ansible Galaxy. However, since Ansible Galaxy is Ansible's official hub for sharing Ansible content, we believe that the majority of roles are hosted on it, making our results representative for the majority of Ansible roles' developers and users.

Conclusion validity concerns the degree to which the conclusions we derived from our data analysis are reasonable. Since our conclusions are mostly based on empirical observations, our work is unlikely to be affected by such threats. However, it is important to mention that our conclusions concern Ansible role smells which are not necessarily bugs, but are indicative of bad practices that may lead to bugs when the role is included in a play.

As a threat to *external validity*, our findings cannot be generalised beyond Ansible roles, i.e., to other Ansible components such as playbooks, or to other Infrastructure as Code languages such as Puppet and Chef. However, it is possible to replicate the design of our study for the aforementioned IaC languages, which we leave as future work.

4.6 Conclusion

Mistakes in infrastructure configuration data are a major cause of infrastructure defects. The intricate details of Ansible's variable and expression semantics may lead developers to unwittingly introduce such defects into their infrastructure code. Moreover, this semantics encumbers program comprehension and therefore introduces maintenance difficulties. Nonetheless, existing research on code smells in Infrastructure as Code has not considered smells originating from such semantics, and existing detection approaches are not able to detect them.

Therefore, we have proposed a catalogue of 6 novel code smells related to the usage of Ansible variables. The smells indicate unsafe reuses of variables, unintended overrides of variables, and variables defined with unnecessarily high precedence. We have combined the Program Dependence Graph representation with a rule engine approach to automatically detect these code smells. This smell detector traverses a PDG and matches logical predicates on each node to identify the smells.

Using the detector, we have conducted an empirical analysis into the prevalence and lifetime of the smells in over 20 000 open-source, reusable Ansible roles. The results show that these smells are becoming increasingly common, and are introduced at a steady pace throughout the entire lifetime of roles. Some smells often co-occur within the same role, suggesting that one smell may cause another. Nonetheless, it may take a long time before smells get fixed, even though anecdotal evidence in the form of fixing commits shows that certain smells may be indicative of infrastructure defects. The proposed smells and the accompanying detector can therefore serve as a valuable asset for Ansible practitioners to spot maintainability and reliability issues in their infrastructure code.

Chapter 5

Detecting Security Weaknesses in Ansible Artefacts

Whereas the previous chapter discussed the *correctness* of Infrastructure-as-Code artefacts and bad practices that may affect them, this chapter instead discusses the *security* of IaC artefacts and the resulting infrastructure. Security-related bad practices can give rise to security vulnerabilities, which may be exploited by malicious actors as an attack vector to abuse enterprise systems hosted on a digital infrastructure. Prior work has identified numerous *security smells*, i.e., recurring patterns indicating potential security vulnerabilities, that can occur in IaC scripts [106, 108, 109, 115]. Various security smell detectors have been developed to detect such smells, including two tools for Ansible, SLAC [108] and GLITCH [115]. These detect various security smells, such as secrets (e.g., passwords) that are embedded in plain text in the infrastructure code (*hardcoded secrets*).

However, these existing detectors rely solely on syntactic analyses (cf. Section 2.3.2), leading to several limitations. The approaches lack awareness of Ansible-specific syntactic constructs, and do not take control-flow and data-flow information into account. These limitations lead them to report false positives, and worse, may lead to vulnerable Ansible code going undetected (i.e., false negatives).

To alleviate these limitations, in this chapter, we propose a novel approach to detect security smells which leverages our Program Dependence Graph (PDG) representation for Ansible (cf. Section 3.3). As this representation includes control-flow and data-flow information, our approach is able to address the latter two limitations. Moreover, as the PDG is constructed from our structural model (cf. Section 3.3.2), which in turn leverages Ansible's own parser (cf. Section 3.2.2), our approach exhibits more syntax awareness. We implement this approach in GASEL (**G**raph-based **A**nsible **Se**curity **L**inter, pronounced "gazelle"), a prototype detector for seven previously-proposed security smells.

To evaluate our approach, we construct an oracle of 243 real-world security smells and measure GASEL's precision and recall. We compare GASEL to SLAC [108] and GLITCH [115], the two state-of-the-art approaches in security smell detection for Ansible code (cf. Sections 2.3 and 5.1), and find that GASEL outperforms the state-of-the-art approaches. However, as our approach is

more expensive to both implement and apply, we investigate whether the higher cost is justified in practice. To this end, we empirically investigate the prevalence of security smells that require control-flow or data-flow information to detect across a large corpus of Ansible scripts. Our findings show that these smells are indeed common in practice, motivating the need for deeper static analysis tools for IaC languages.

The remainder of this chapter is structured as follows. In Section 5.1, we survey the academic literature for state-of-the-art approaches performing security smell detection in Infrastructure as Code, and present the catalogue of security smells considered in our work. Then, Section 5.2 introduces motivating examples that highlight the need to transcend syntactic analysis for security smell detection. Section 5.3 addresses this need by introducing our smell detection approach based on PDGs, and the GASEL prototype that implements it. We employ GASEL in Section 5.4 to empirically study security smells on a large scale. Section 5.5 discusses the empirical study's results and implications, after which Section 5.6 concludes.

A replication package containing our prototypes, data, and analysis scripts used in this chapter is available at `https://doi.org/10.6084/m9.figshare.21929856`.

5.1 State of the Art

In this section, we review the literature on security smell detection for Infrastructure as Code. We focus on two aspects, first on the smell detection approaches (Section 5.1.1) and afterwards summarising a catalogue of security smells for configuration management languages (Section 5.1.2).

5.1.1 Security Smell Detectors

The earliest work on security smells in infrastructure code is by Rahman et al. [106], who investigate security weaknesses in Puppet scripts. They devise a catalogue of several security smells, such as hardcoded secrets and missing integrity checks on downloaded executable code, and develop SLIC, a tool to detect them. Rahman et al. [108] replicate this work for Ansible and Chef, developing a similar tool named SLAC, and later for Kubernetes as well [109].

For Ansible, SLAC parses the Ansible YAML code into dictionaries and lists. It traverses this representation and checks pairs of dictionary keys and values in search of smells according to a set of detection rules combined with a set of string patterns. For instance, to detect a hardcoded secret, SLAC searches for keys denoting a secret (e.g., keys named "password") with an associated value that is a literal string.

SLIC and SLAC implement the same detection rules for the three languages individually. This redundancy led Saavedra and Ferreira [115] to create GLITCH, a polyglot detection tool for Puppet, Chef, and Ansible, which they later extended to Terraform and Docker [116]. Although the detection rules are

largely identical to Rahman et al.'s, GLITCH only implements a single version operating on an intermediate representation. Saavedra and Ferreira also make numerous improvements to the string patterns proposed by Rahman et al., enabling GLITCH to outperform SLIC and SLAC. Moreover, GLITCH implements all applicable rules for all languages, whereas SLIC and SLAC only detect a subset of the security smells for each individual language. For Ansible, GLITCH's intermediate representation is a lightweight abstraction over the dictionaries and lists obtained by YAML parsing. Ansible-specific concepts, such as tasks, variables, and expressions, can be tagged in the representation.

Reis et al. [113] replicate SLIC's evaluation and find that its precision is substantially lower than originally reported after validation with code owners, dropping from 99% to 28%. Armed with practitioners' feedback, they develop InfraSecure, a smell detector for Puppet which outperforms SLIC. To this end, they implement several improvements, such as introducing more syntax awareness and adjusting the detection rules to reduce false positives.

Finally, Rahman and Parnin [105] adapt SLIC to identify which Puppet resources are impacted by security smells. To this end, they construct a data dependence graph of a Puppet manifest (cf. Sections 2.3.2 and 3.1.2) and propagate a security smell detected in one resource to other resources by traversing along the data dependence edges. They use these propagated defects to empirically study the impact of weaknesses across resources in an IaC script. Rahman et al. [101] later replicate this approach and the empirical study for Ansible. However, the detection approaches employed in both studies remain the same as in the original SLIC and SLAC tools, operating at a purely syntactic level. Therefore, the new tools are still unable to detect smells that require control-flow or data-flow information.

5.1.2 Security Smell Catalogue

Prior work has proposed various catalogues of smells, many inspired by one another and accompanying a specific detection tool. Of particular interest to our work is the catalogue described by Saavedra and Ferreira [115], who combined the smell catalogues of Rahman et al. for Puppet [106] and Ansible and Chef [108]. The catalogue comprises 8 security smells that are applicable to Ansible, described below.

Admin by default (ADMIN) Using administrator privileges by default may be indicative of a violation of the principle of least privilege. Examples include specifying a user with administrator privileges, running a service as the root user, or accessing a resource with elevated permissions.

Empty password (EMPTY) Configured passwords must not be empty, as empty passwords are trivial to crack.

Hardcoded secret (SECRET) Hardcoding sensitive information such as passwords or private keys into code can lead to severe vulnerabilities when

the code is published online, either accidentally into open-source repositories or maliciously through leaks. Secrets should therefore be stored in so-called *vaults* [100].

HTTP without SSL/TLS (HTTPS) Omitting encryption protocols when communicating over HTTP allows an attacker to intercept or modify communications through man-in-the-middle attacks. Communication should therefore be encrypted with SSL or TLS to prevent tampering and eavesdropping.

Missing integrity check (INTEGRITY) When downloading code archives or executables, the downloaded file's integrity should be checked using cryptographic hashes. Unwanted modifications will otherwise go unnoticed, enabling attackers to perform software supply chain attacks by tampering with the code executed on infrastructure machines.

Suspicious comment (COMMENT) Developers may embed comments in their infrastructure code that may reveal flaws or shortcomings in the system. These often mention certain keywords, such as "`TODO`" or "`FIXME`". When these comments concern the system's security, they are indicative of security flaws, and may also provide information on how the flaw can be exploited.

Unrestricted IP address (IP) In network configurations, using the IP address `0.0.0.0` indicates that the server should accept connections from any network interface, including remote connections. Missing IP address filters facilitate denial of service attacks.

Weak crypto algorithm (CRYPTO) Some cryptographic algorithms have been shown to be insecure, as flaws can be used to break their security guarantees. For instance, using hashing algorithms like SHA-1 or MD5 to hash passwords or perform file integrity checks is insecure, as these algorithms are prone to collision attacks. Such algorithms should therefore no longer be used.

5.2 Motivating Examples

The motivating examples depicted in Listings 5.1 and 5.2 highlight the 3 major limitations exhibited by SLAC and GLITCH.

5.2.1 Lack of Ansible Syntax Awareness

As described in Section 3.2, Ansible features extended syntax that is not handled by standard YAML parsing. The most notable example of this is the inline task module arguments syntax (cf. Section 2.2.1). YAML parsing therefore does not suffice to create the key-value pairs required for existing tools to check these arguments for smells. For example, existing tools cannot properly parse the task arguments in Listing 5.1, and thus miss the *HTTP without SSL/TLS* and *missing integrity check* smells, as they cannot identify the

```
1 - name: Download nginx-{{ version }}.tar.gz
2   get_url:
3     url=http://nginx.org/download/nginx-{{ version }}.tar.gz
4     dest=/usr/local/src/nginx-{{ version }}.tar.gz
```

LISTING 5.1: Task adapted from `personium/Ansible` exhibiting *HTTP without SSL/TLS* and *missing integrity check* smells requiring Ansible-specific syntax awareness to detect.

```
1  - hosts: ...
2    roles:
3      - role: overdrive3000.percona
4        root_password: _password_
5
6  # In overdrive3000/percona role
7  - name: Update MySQL root password
8    mysql_user:
9      name: root
10     password: '{{ root_password }}'
```

LISTING 5.2: Play adapted from `CenturyLinkCloud/clc-ansible-module` exhibiting a *hardcoded secret* smell requiring control-flow and data-flow information to detect.

URL. Moreover, since existing approaches do not fully support expressions, they may miss the smells even when standard YAML notation is used.

5.2.2 Lack of Data-Flow Information

Existing approaches do not take the flow of data within an Ansible script into account. This can lead to false negatives when smells feature indirection through variables and expressions. In Listing 5.2, for instance, a password is specified as an expression (line 10) that refers to a variable (line 4). The latter is initialised with a literal, thus forming a hardcoded secret. Lacking data-flow information also causes tools to report false positives for smells that cannot be harmful in practice, e.g., due to dead code or unused variables.

5.2.3 Lack of Control-Flow Information

Recall from Section 2.2.1 that Ansible scripts can be structured as separate YAML files which are dynamically included at run time. Moreover, they can dynamically include Ansible roles originating from third parties, as depicted in Listing 5.2, where a play (lines 1–4) dynamically includes a role from a third-party dependency. Existing approaches do not inspect the control flow

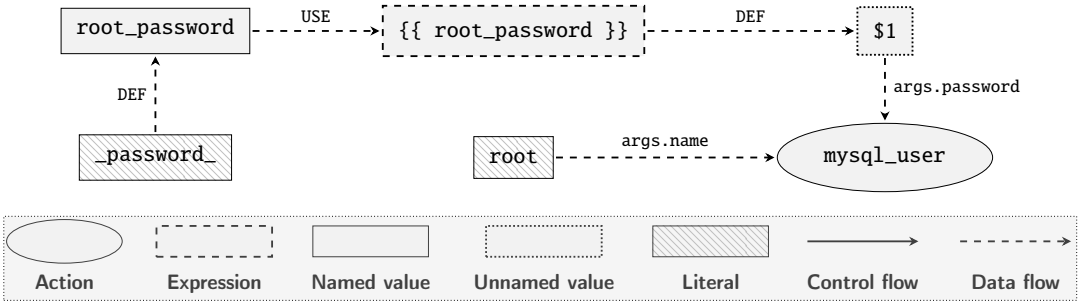

FIGURE 5.1: Program Dependence Graph of the example depicted in Listing 5.2.

within Ansible scripts, and must instead rely on scanning all YAML files and assuming they contain Ansible code. However, not every YAML file contains relevant Ansible code. The tools therefore report false positives, such as for test code which developers often consider harmless [108], or for non-Ansible YAML files.

Furthermore, dynamic inclusion can cause the aforementioned data-flow indirection to cross files, where a variable is defined in one file and used in another. For instance, in Listing 5.2, the hardcoded secret smell is actually spread across different files, with the password being defined in one file (line 4) but used in another (line 10). Due to Ansible's variable and expression semantics (cf. Section 2.2.4), detecting such indirect smells requires accurately tracking both control and data flow, and thus necessitates a whole-project analysis.

5.3 Graph-based Security Smell Detection

To address the above limitations, we introduce an approach to detect the previously-proposed security smells in Ansible using graph queries on the Program Dependence Graph representation (cf. Section 3.3). Our approach addresses the first limitation because the PDGs are built from our structural model, which is aware of Ansible's extended syntax. Furthermore, the data-flow information embedded in the PDG enables our approach to overcome the second limitation. Finally, as our PDGs are built using a whole-program analysis, they contain information about several files as well as third-party code, addressing the third limitation of state-of-the-art detectors.

To illustrate, Figure 5.1 depicts the PDG of Listing 5.2, showing that the PDG comprises the code from all involved files. Moreover, it makes clear that there exists a data-flow path from the literal value to the password argument of the task, indicating a *hardcoded secret* smell with data-flow indirection.

Our approach operates in two phases. First, it builds a PDG for an "entry point", either an Ansible playbook or an Ansible role, using the PDG builder described in Section 3.3.2. Then, it runs Cypher graph queries on each entry point's PDG. We devise 7 such Cypher queries, each detecting the instances of one type of security smell that may be present in an Ansible playbook or role.

TABLE 5.1: Detection rules for security smells. The symbol n denotes any node, l, v, and t denote literal, variable, and task action nodes, respectively, and a denotes argument edge labels. The notation $n_1 \cdots\xrightarrow{a} n_2$ denotes the existence of a data-flow path of arbitrary length between n_1 and n_2, ending with an edge labelled a.

Smell type	**Query description**
ADMIN	$l \cdots\xrightarrow{a} t \wedge isAdmin(l) \wedge isUser(a)$
EMPTY	$\big((l \cdots\xrightarrow{a} t \wedge isPassword(a)) \vee (l \cdots\longrightarrow v \cdots\longrightarrow t \wedge isPassword(v))\big)$ $\wedge\, isEmpty(l)$
SECRET	$\big((l \cdots\xrightarrow{a} t \wedge isSecret(a) \wedge \neg isSecretWhitelist(a)) \vee$ $(l \cdots\longrightarrow v \cdots\longrightarrow t \wedge isSecret(v) \wedge \neg isSecretWhitelist(v))\big)$ $\wedge\, \neg isEmpty(l)$
HTTPS	$n \cdots\longrightarrow t \wedge (isLiteral(n) \vee isExpression(n)) \wedge isHTTP(n)$ $\wedge\, \neg\big(isHTTPWhitelist(n) \vee (l \cdots\longrightarrow n \wedge isHTTPWhitelist(l))\big)$
INTEGRITY	$\big(l_1 \cdots\longrightarrow t \wedge isDownload(l_1) \wedge \neg(n \cdots\xrightarrow{a_1} t \wedge isChecksum(a_1))\big) \vee$ $\big(l_2 \cdots\xrightarrow{a_2} t \wedge ((isCheckFlag(a_2) \wedge \neg l_2) \vee (isNoCheckFlag(a_2) \wedge l_2))\big)$
IP	$n \cdots\longrightarrow t \wedge (isLiteral(n) \vee isExpression(n)) \wedge isBadIP(n)$
CRYPTO	$n \cdots\longrightarrow t \wedge (isLiteral(n) \vee isExpression(n)) \wedge isWeakCrypto(n)$

We instantiated this approach into GASEL, a prototype detector implemented in Python using RedisGraph as an in-memory graph database that answers the Cypher queries.

Cypher Queries

We design a Cypher query for 7 of the 8 smells supported by GLITCH (see Section 5.1.2). We omit the *suspicious comment* smell, as comments are not represented in the PDG and control-flow and data-flow information will not improve its detection.

To design the graph queries, we took initial inspiration from the matching rules of the GLITCH tool. However, our queries match paths in the graph rather than keys and values of a tree-based representation. A major advantage is that our queries can account for data-flow indirection in the code by matching variable-length definition-use chains. As a result, our queries report the indirectly hardcoded password exemplified in Listing 5.2, thus addressing the limitation described in Section 5.2.2.

Table 5.1 summarises our graph queries for security smell detection. Listing 5.3 depicts a simplified example of a corresponding Cypher query. In the table, the operation $n_1 \cdots\xrightarrow{a} n_2$ finds definition-use paths of arbitrary length between nodes n_1 and n_2, where the final edge in the path is denoted by a, which may

Table 5.2: String patterns used in graph queries.

Function	String pattern
isAdmin	`admin, root`
isUser	`user, role, uname, login, ...`
isPassword	`pass, pwd`
isSecret	`pass, pwd, token, secret, ssh.*key, ...`
isSecretWhitelist	`generate, update`
isHTTP	`http://`
isHTTPWhitelist	`localhost, 127.0.0.1`
isDownload	`http.+.tar.gz, http.+.dmg, http.+.rpm, ...`
isChecksum	`checksum, cksum`
isCheckFlag	`gpg_check, check_sha`
isNoCheckFlag	`disable_gpg_check`
isBadIP	`0.0.0.0`
isWeakCrypto	`md5, sha1, arcfour`

```
1 MATCH (l:Literal)-[:DEF|USE*0..]->()-[a:ARG]->(t:Action) // l ···a→ t
2 WHERE
3     (a.value CONTAINS "user" OR a.value CONTAINS "role") // isUser(a)
4     AND (l.value = "admin" OR l.value = "root") // isAdmin(l)
5 RETURN l.location;
```

Listing 5.3: Simplified graph query for *admin by default* smells.

be omitted if the edge label is irrelevant. Functions *isExpression* and *isLiteral* check the node type of their argument, while functions such as *isPassword* or *isUser* check the label of a given node or edge. Similarly to SLAC and GLITCH, the latter functions use string patterns, summarised in Table 5.2.

Our string patterns are inspired by those of GLITCH, which have previously been shown to outperform those of SLAC, but we slightly refined the patterns using our Ansible domain-specific knowledge. For instance, we add a whitelist to the *hardcoded secret* query to account for common string-valued arguments that are used as flags rather than secrets (e.g., `update_password`, which specifies a boolean value indicating whether a password must be updated). Importantly, for the *hardcoded secret* smell, we do not consider usernames to be secret, following the practitioner feedback reported by Reis et al. [113].

5.4 Security Smells in Practice

This section presents our empirical study into the prevalence of security smells in real-world Ansible projects. We investigate the following research questions.

- *RQ_1: How accurate is our security smell detector?* This RQ serves as the evaluation of our approach. We construct an oracle of real-world security smells, use it to determine precision and recall of our approach, and compare it to two state-of-the-art security smell detectors for Ansible, namely SLAC [108] and GLITCH [115].
- *RQ_2: How prevalent are security smells in open-source Ansible codebases?* In this RQ, we investigate how often security smells of different types are manifested in Ansible repositories, entry points, and files. The results will provide insights into the prevalence of security smell types in practice.
- *RQ_3: How often do security smells cross file boundaries?* In this RQ, we investigate the instances of security smells detected in RQ_2 which require a whole-program analysis to detect. Such instances involve multiple Ansible files (e.g., a hardcoded password variable defined in one file but used in another) or are partly or wholly situated in third-party dependencies, and may be undetectable to single-file analyses. Answering this RQ provides insights into the importance of whole-program analyses in security smell detection.
- *RQ_4: How prevalent is data-flow indirection in security smells?* Our final RQ investigates how frequently smell instances involve data-flow indirection through the use of variables and expressions. Similar to before, the answer to this question will determine the need to account for data flow to accurately detect security smells in practice.

5.4.1 Dataset Collection

To answer these questions, we need a dataset of Ansible repositories. Whereas in the previous chapter, we used a dataset of reusable Ansible roles, i.e., "libraries", we now expand our scope to also include Ansible playbooks, i.e., the end user's scripts. However, identifying openly-accessible playbooks is difficult, as they are not hosted on dedicated indexers such as Ansible Galaxy. Moreover, as Ansible files can have arbitrary names and do not carry a unique file extension, instead using the YAML extensions, searching based on file names is impractical. Instead, we start from the dataset previously collected by Saavedra and Ferreira [115]. They collected a list of 681 repositories from GitHub and applied various filtering criteria focused on development characteristics (number of contributors, number of commits) to discard irrelevant projects. However, we argue that their dataset collection procedure may have missed several relevant, frequently-used repositories, while simultaneously retaining irrelevant projects. Therefore, we aim to augment this initial dataset in two ways.

First, using our dataset of Ansible roles previously collected from the Ansible Galaxy registry in Section 4.4.1, we add popular repositories that were missed in Saavedra and Ferreira's [115] dataset. However, rather than using all roles, we aim to curate the dataset to omit low-quality or rarely-used Ansible roles.

Table 5.3: Dataset statistics.

Attribute	Original	Extension	Total
# repositories	448	524	972
# owners	285	196	449
# YAML files	74 862	8 932	83 794
# non-test YAML files	64 311	6 449	70 760
# playbooks	7 744	219	7 963
# roles	7 490	527	8 017

Therefore, from the dataset of Ansible roles, we identify the most popular repositories that cumulatively represent 95% of all role downloads in the Ansible Galaxy ecosystem. We found a total of 710 repositories, from which we omit 64 that are already present in the original dataset, and a further 34 which are forks. Thus, we obtain 612 new relevant repositories.

Second, we apply additional criteria to remove repositories that are rarely used and are thus not representative. To this end, we employ GitHub's "stars" metric, which counts the number of people who have "starred" a repository. We remove all repositories that have no stars on GitHub, which indicates a lack of popularity. This removes 272 repositories from the dataset, 199 of which originate from the original dataset by Saavedra and Ferreira [115]. Moreover, we remove 49 "hidden" forks, 34 of which originate from the original dataset. These are repositories that are not marked as forks on GitHub but share an initial commit with another repository in the dataset. Such repositories may share a lot of code with another repository in the dataset, and their inclusion may skew our results. For each group of forks, we retain the most popular repository as indicated by the number of stars.

We obtain a final dataset of 972 Ansible repositories comprising 15 980 entry points (playbooks or roles). A summary of this dataset is depicted in Table 5.3.

5.4.2 RQ_1: How accurate is our security smell detector?

Research method To calculate precision and recall and compare to the state of the art, we require a set of true security smells in Ansible scripts. Although prior work provides such a manually annotated oracle [108], it does not serve our needs for three reasons. First, the oracle considers individual files, whereas our approach performs a whole-project analysis. Second, for several security smells, the oracle contains very few or no true positives, which may lead to unrepresentative results. Third, the oracle is constructed of files belonging to a single project, which hinders generalisability.

Instead, we create a new ground-truth oracle by sampling the results of the three detectors on a corpus of Ansible projects and by pooling [74] the true positive reports. Specifically, we first run the three detectors on the entire corpus of Ansible repositories. We run GASEL on each playbook and role

in each repository and rely on its ability to resolve and subsequently scan included files. We configure it to search for third-party role dependencies in a search path containing *all* roles in the Ansible Galaxy dataset from Chapter 4. Then, we run SLAC and GLITCH on each repository in the corpus using the configurations from their respective replication packages. We filter out all reports of suspicious comments and hardcoded usernames, since these are not implemented in GASEL, as motivated in Section 5.3.

Next, we randomly select a sample of reports for manual review. However, since SLAC does not report line numbers, and line numbers reported by GASEL may differ slightly from those reported by GLITCH, we cannot automatically relate the reported smells. Instead, we sample entire files and manually review all reported smells for those files.

To obtain a varied set of files that is representative of all detectors, we first group the files into three categories for each security smell: files for which GASEL reports the same number of smells as another tool, files for which GASEL reports more smells, and files for which GASEL reports fewer smells. The aim is to review both smells that are commonly found, and smells that are missed by at least one tool. As noted above, GASEL relies on resolving dynamically included files to perform a whole-project analysis, but resolving such files statically is not always possible. As such, it may occasionally overlook files that were scanned by SLAC and GLITCH, which do not rely on control-flow information and simply scan all YAML files. Conversely, SLAC and GLITCH may mistakenly scan irrelevant files, such as non-Ansible YAML files, which would degrade their results. To reduce this bias, we decided to only consider the files that were scanned by all three tools. Subsequently, we randomly select 10 files of each category for each security smell, leading to a total of 210 files to be manually reviewed. Note that each file can contain multiple security smells of the same type.

We then manually label each sampled smell as a true or false positive. We consider smells to be false positives if they cannot cause a security weakness in the project (e.g., a reported hardcoded secret that is not a secret, or a variable containing a smell while that variable is never used). Through the resulting oracle, we will be able to calculate each tool's precision and recall. Note that the obtained recall values will be an approximation, since the oracle will lack smells that are missed by all detectors.

Results In total, we manually reviewed 390 unique potential smells from 662 reports by the three detectors. We find 243 true positives, ranging between 11 and 64 per smell type. A summary of the manual investigation and the oracle is provided in Table 5.4.

Table 5.5 depicts the precision and recall for the three detectors on the oracle set of security smells. Note again that the recall values are an approximation, since the oracle does not contain smells missed by all detectors.

GASEL achieves the highest recall for 6 of the 7 smells, by an often substantial margin. For all smells, GASEL finds more than 75% of their instances in the

Table 5.4: Oracle construction results.

Smell type	# checked reports	# unique reports	# unique true positives
Admin by default	96	65	64
Empty password	74	37	15
HTTP without SSL/TLS	145	99	35
Hardcoded secret	65	42	11
Missing integrity check	60	37	27
Unrestricted IP address	143	61	47
Weak crypto algorithm	79	49	44

Table 5.5: Precision (P) and recall (R) for GASEL, SLAC, and GLITCH on the oracle dataset. Highest precision and recall for each smell are marked in bold.

	GASEL		GLITCH		SLAC	
Smell type	**% P**	**% R**	**% P**	**% R**	**% P**	**% R**
ADMIN	98.11	**81.25**	**100.00**	67.19	*N/A*	*N/A*
EMPTY	**44.44**	**80.00**	42.86	60.00	30.77	53.33
HTTPS	**100.00**	**88.57**	54.84	48.57	22.89	54.29
SECRET	45.45	**90.91**	**56.25**	81.82	33.33	81.82
INTEGRITY	**96.15**	**92.59**	50.00	33.33	75.00	44.44
IP	76.60	76.60	**82.98**	82.98	81.63	**85.11**
CRYPTO	**97.67**	**95.45**	86.11	70.45	*N/A*	*N/A*

sample set. GASEL also achieves the highest precision for 4 of the 7 security smells, and only a slightly lower precision than GLITCH for another.

We further observe that for 4 smells, namely *empty password*, *HTTP without SSL/TLS*, *missing integrity check*, and *weak crypto algorithm*, our tool achieves both the highest precision and recall of all detectors. For *HTTP without SSL/TLS* and *missing integrity check*, our tool substantially outperforms SLAC and GLITCH. For the *admin by default* and *hardcoded secret* smells, the improvement in recall is paired with lower precision, possibly indicating a trade-off between precision and recall. Finally, we note a substantial decrease in both precision and recall for the *unrestricted IP address* smell, where GASEL performs worst.

Apart from precision and recall, a secondary yet important concern in security smell detection is the time taken to scan a project. Unsurprisingly, because GASEL needs to perform more in-depth analysis, it takes more time than the other tools. In our experiments, SLAC and GLITCH took 8 and 22 minutes respectively to scan the entire dataset, while GASEL took slightly over 2 hours. Nonetheless, GASEL's mean running time is around 220ms to scan an entire repository and around 65ms to scan a single playbook or role, which is acceptable. However, large projects may form large outliers, with some repositories taking multiple minutes, upwards of 40 minutes for a single repository. For such projects, GASEL may need to scan the same Ansible file many times if the file is included multiple times in the repository (e.g., a role included by multiple playbooks). Another factor is the use of a graph database in GASEL's prototype implementation. We observe that a large portion of time is spent merely importing the built PDG into this database. We theorise that performance tuning of the database may aid in improving the performance on large projects and consequently, large PDGs.

Answer to RQ_1: Our approach achieves consistently high recall (above 75% for all smells). Its precision is also often high (>95% for 4 smells), except for *empty password* and *hardcoded secret* smells. It achieves the highest precision and recall for 4 and 6 smell types, respectively, but performs considerably worse for *unrestricted IP address*.

5.4.3 RQ_2: How prevalent are security smells in open-source Ansible codebases?

Research method We run GASEL on the entire corpus of repositories according to the same setup as detailed in RQ_1. However, we additionally instruct it to produce a *sink* location for each smell, i.e., the location of the task that it affects. This is different from the smell's own location, which we shall from now on refer to as the *source* location, in part because of control-flow and data-flow indirection.

Note that since GASEL scans each entry point separately, it may report duplicates when smells are reachable from multiple entry points in a repository, e.g., because both entry points include a common file. Since duplicate reports

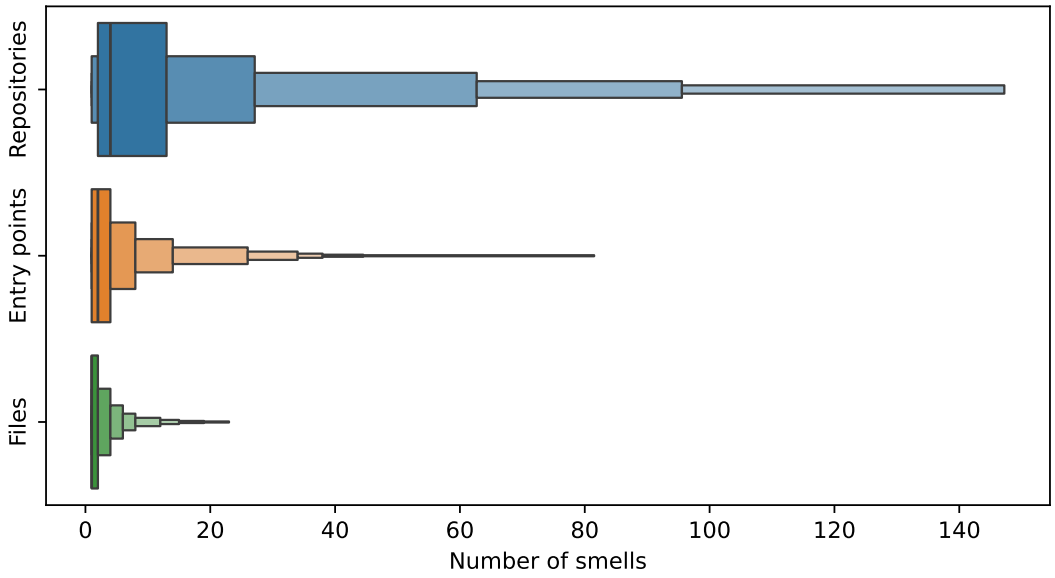

FIGURE 5.2: Letter-value plots depicting the distribution of the number of smells per repository, entry point and sink file. Outliers are omitted for space considerations.

would skew our results, we eliminate them by not considering the entry point through which the smells are reported, except when investigating the proportion of entry points exhibiting smells.

Results GASEL detected a total of 7 933 unique security smells impacting 472 repositories (48.56% of our corpus) and 3 457 entry points (21.63% of the corpus). The mean and median number of affected entry points per repository are 7.32 and 1, respectively, while the mean and median number of smells per affected entry point are 4.42 and 2.

These smells are spread across 3 145 source files, impacting a total of 3 613 sink files, indicating that certain smell sources are used by multiple sinks. The distribution of the number of smells per repository, entry point, and sink file are summarised in Figure 5.2. We observe a mean and median number of smells per sink file of 2.2 and 1, respectively. For repositories, we find that the mean and median number of smells are 16.8 and 4. The high difference between the two suggests the presence of outliers with many smells, which can be seen in the letter-value plot. Indeed, we found repositories with as many as 802 smells. Similarly, the maximum number of smells in a single entry point is 247.

Table 5.6 summarises the number of smells, affected files, and affected repositories per smell type. We find that *hardcoded secret* is the most common security smell across all three metrics. However, note that our approach achieves low precision for this smell type (RQ_1), and its prevalence is thus likely an over-approximation. In terms of number of smells, *hardcoded secret* is closely followed by *admin by default*, yet in terms of number of affected files and repositories, *missing integrity check* ranks second. This may suggest that

TABLE 5.6: Number of smells, affected repositories and sink files grouped by smell type. Percentages of affected repositories and sink files are relative to the total number of *affected* repositories and files.

	Smells		Affected sinks		Affected repos	
Smell type	**#**	**%**	**#**	**%**	**#**	**%**
SECRET	2 155	27.17	1 235	34.18	211	44.70
ADMIN	2 043	25.75	766	21.20	190	40.25
HTTPS	1 369	17.26	765	21.17	198	41.95
INTEGRITY	1 084	13.66	803	22.23	200	42.37
CRYPTO	577	7.27	234	6.48	94	19.92
IP	418	5.27	289	8.00	120	25.42
EMPTY	287	3.62	163	4.51	65	13.77

repositories or files often contain multiple *admin by default* smells while *missing integrity check* more often occurs alone. The table also shows that smell types are not evenly distributed across all repositories, since the highest number of repositories impacted by a smell type (211) is less than half of all affected repositories (472).

Finally, Figure 5.3 depicts the distribution of the number of smells per repository and sink file, grouped by smell type. Note that for each group, we only focus on those repositories or files that exhibit at least one smell of the given type, so the minimum number in each is 1. We observe that the median number of smells per entry point is 1 for all smell types, whereas the median for repositories ranges between 2 and 4 per smell type. This figure also confirms that repositories or files frequently have multiple *admin by default* instances, explaining the observation described above.

Answer to RQ_2: 49% and 22% of the repositories and entry points in our dataset are affected by 7 933 unique security smells. *Hardcoded secret* is the most prevalent, followed by *admin by default* and *missing integrity check*.

5.4.4 RQ_3: How often do security smells cross file boundaries?

Research method To study control-flow indirection, we build upon the instances collected in RQ_2 and focus on those whose source file differs from their sink file, indicating a control-flow indirection through file inclusion. We also investigate smells of which the source or sink file is located in third-party code (role dependencies), and thus get included into the client code indirectly.

Results We observe that 2 594 (32.7%) instances have a source file different from the sink file and thus cross file boundaries. The proportion of such

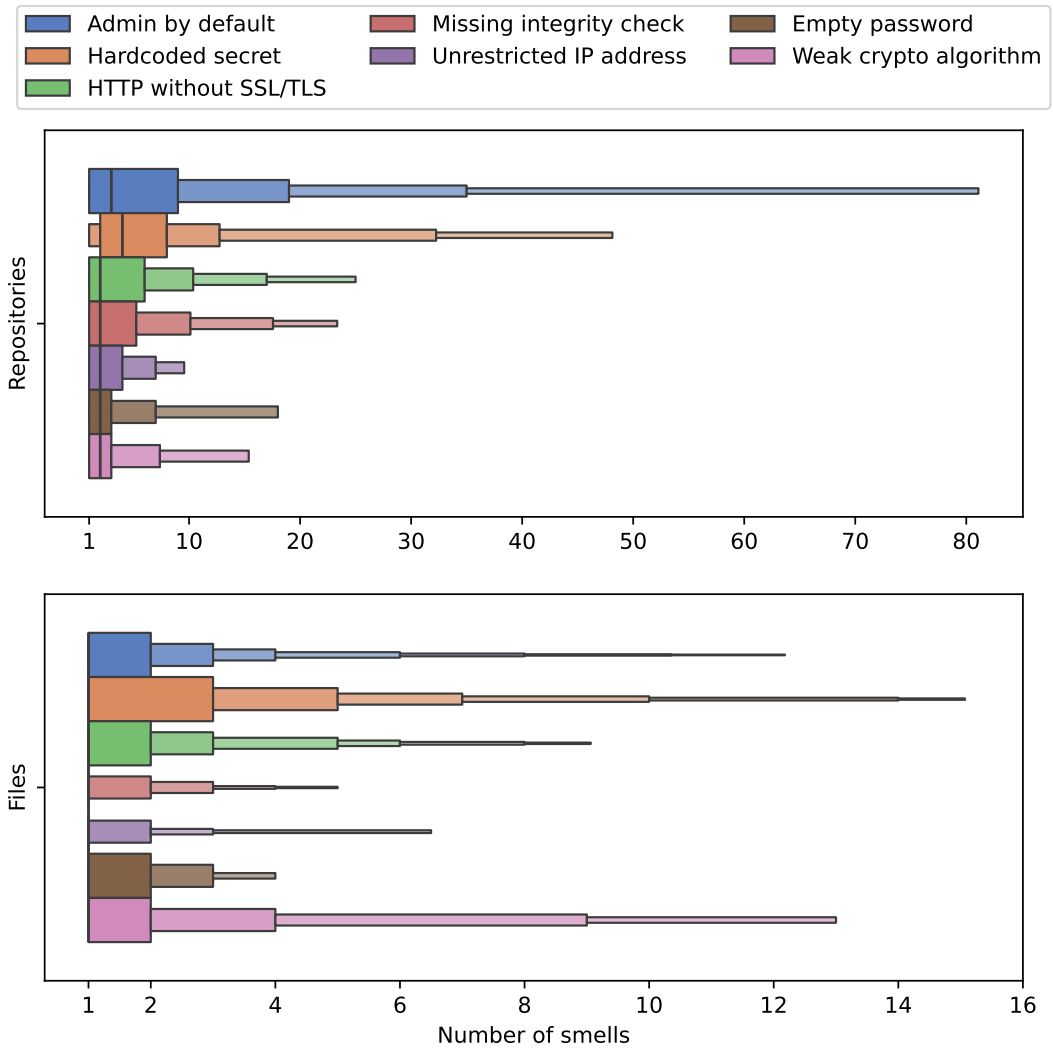

FIGURE 5.3: Letter-value plots depicting the distribution of the number of smells per repository and sink file, grouped by smell type. Outliers are omitted for space considerations.

instances is the highest for *empty password* (51.92%) and *weak crypto algorithm* (51.13%) smells. Conversely, *admin by default* (21.34%) and *unrestricted IP address* (23.68%) exhibit the lowest proportion of such instances. For all smell types, more than 20% of their instances affect tasks defined in different files.

Moreover, we find that 510 smells (6.43%) are situated entirely within third-party code. The majority of these are *admin by default* (150) and *HTTP without SSL/TLS* (109) smells, whereas *weak crypto algorithm* (12) and *empty password* (13) represent the lowest number of smells. The smell type with the highest proportion of smells in third-party code is *unrestricted IP address* (15.55%), while *weak crypto algorithm* has the lowest (2.08%). Each smell type has instances situated in third-party roles.

Finally, GASEL detected 20 smells (0.25%) that cross the boundaries of first-party and third-party code. Specifically, we find 9 *missing integrity check*, 8

hardcoded secret, and 3 *HTTP without SSL/TLS* smells where a variable defined in first-party code is used by a task in third-party role code.

Answer to RQ$_3$: 33% of smell instances involve file inclusion, while 6.5% partly or fully involve third-party code. While *admin by default* and *unrestricted IP address* exhibit the lowest proportion of instances crossing file boundaries, they comprise large numbers of instances involving third-party code. Conversely, *weak crypto algorithm* and *empty password* exhibit the highest proportion of instances crossing file boundaries, yet rarely involve third-party code.

5.4.5 RQ$_4$: How prevalent is data-flow indirection in security smells?

Research method We again run GASEL on the entire corpus of repositories using a similar setup as before. However, we now instruct it to produce a "data-flow indirection level" for each smell, which resembles the length of the definition-use chain between a smell's source and sink, counted as the number of variables in this chain. For instance, indirection level 2 indicates that the sink refers to a variable which in turn depends on another variable, the latter constituting the source of the smell. Indirection level 0 indicates that the smell occurs directly as a task argument, without variables. For example, the *hardcoded secret* smell in Listing 5.2 has an indirection level of 1.

Results We find that 55.5% (4 402) of the detected smells involve some level of indirection through variables and expressions. Table 5.7 depicts the number of smells, repositories, and sink files, grouped by the smell's indirection level. We observe that the majority of indirect smells only use one level of indirection, with higher indirection levels becoming increasingly rare. The highest indirection level observed is 6, which occurred only 7 times. However, a manual investigation of these smells with high indirection levels suggests that many are false positives.

Figure 5.4 depicts a heatmap of the proportion of indirection levels per smell type. We observe that a majority of *admin by default* (77%) and *unrestricted IP address* (66%) instances do not contain data-flow indirection. Conversely, the 5 other security smells contain data-flow indirection more often than not. For *empty password*, we observe that the vast majority (85%) of its instances exhibit one level of indirection. We further find that large proportions of *HTTP without SSL/TLS* (31%) and *weak crypto algorithm* (28%) instances exhibit two or more levels of indirection.

TABLE 5.7: Number of smells, affected repositories and sink files grouped by smell data-flow indirection level. Percentages of affected repositories and sink files are relative to the total number of *affected* repositories and files.

Indirection level	Smells		Affected sinks		Affected repos	
	#	%	#	%	#	%
0	3 531	44.51	1 759	50.40	326	69.07
1	3 227	40.68	1 565	44.84	304	64.41
2	957	12.06	456	13.07	134	28.39
3	181	2.28	70	2.01	36	7.63
4	17	0.21	11	0.32	10	2.12
5	13	0.16	7	0.20	7	1.48
6	7	0.09	3	0.09	2	0.42

Answer to RQ$_4$: Over 55% of security smells contain data-flow indirection. While *admin by default* and *unrestricted IP address* rarely contain data-flow indirection, the remaining types more often than not exhibit data-flow indirection. Data-flow indirection mostly involves one variable, while indirection through more than 3 variables is rare.

5.5 Discussion

In this section, we discuss the practical implications of our results and how they impact the prevalence of security smells reported by prior work. We furthermore discuss limitations of our approach and potential directions for future work. We start by investigating the causes for differences in the smells reported by the studied detectors.

5.5.1 Causes for Differences in Detector Reports

While evaluating GASEL's precision and recall, we determined the root causes for GASEL finding smells where other detectors did not, and vice versa.

Syntax Awareness

Awareness of Ansible syntax is a major reason for new true positives found by our approach. The ability to correctly parse the inline task module arguments syntax (see Section 5.2.1) contributes substantially to the improved recall for *admin by default* and *HTTP without SSL/TLS* smells. Among others, the smell exemplified in Listing 5.1 was found by GASEL but missed by other tools. Furthermore, awareness of the `register` directive enabled our approach to avoid numerous false positives of *weak crypto algorithm*. This directive, whose

FIGURE 5.4: Heatmap showing the proportion of smells of each group (by type) and its indirection level.

value is a variable name used to store the outcome of a task, led GLITCH to falsely report instances because of the use of `md5` in the variable name.[1]

Data-flow Information

Support for expressions and indirection uncovered 8 new instances of *missing integrity check* smells. For instance, the real-world example depicted in Listing 5.4 was not detected by other tools due to the indirection between the task and the variable containing the source URL. Moreover, GASEL uncovered one new instance each for *admin by default*, *hardcoded secret*, and *weak crypto algorithm*, among which the motivational example depicted in Listing 5.2. Apart from new true positives, it also avoided false positives (i.e., new true negatives) for *HTTP without SSL/TLS*, where `localhost` is used indirectly through an expression to construct a URL, as exemplified in Listing 5.5. Note that this example is heavily simplified. In reality, both variable definitions were in separate files and thus also required a whole-program analysis. Finally, data-flow information also allowed GASEL to avoid a handful of false positives caused by variables that are not used in a project.

However, the inclusion of data-flow information also caused new false positives to be reported by GASEL. Specifically, it reported multiple instances of paths to secret files (e.g., certificates and key files) that are constructed using expressions, exemplified in Listing 5.6. Although the content of such files is secret, the

[1] Note that the affected tasks often did exhibit the use of weak crypto algorithms, which were caught by both tools, but GLITCH reported these twice.

```yaml
# Contents of defaults/main.yaml
webapp_operator_tmp: '/tmp/webapp-operator'
webapp_operator_release_tag: '0.0.63-workshop-1'
webapp_operator_resources:
  'https://github.com/.../v{{ webapp_operator_release_tag }}.zip'

# Contents of tasks/provision-webapp.yml
- name: Download example files
  unarchive:
      src: '{{ webapp_operator_resources }}'
      dest: '{{ webapp_operator_tmp }}'
      remote_src: yes
```

Listing 5.4: Simplified example of a *missing integrity check* smell that uses data-flow indirection, adapted from `redhat-cop/agnosticd`.

```yaml
- ini_file:
    path: "{{ opensds_conf_file }}"
    section: osdslet
    option: "prometheus_push_gateway_url"
    value: "{{ prometheus_push_gateway_url }}"
  vars:
    prometheus_push_gateway_url: 'http://{{ host_ip }}:9091'
    host_ip: 127.0.0.1
```

Listing 5.5: Simplified example of a false *HTTP without SSL/TLS* smell avoided using data flow, adapted from `sodafoundation/installer`.

reported values only contain the path and are thus false positives. Further improvements to the string patterns may aid in avoiding these false positives.

Control-flow and Contextual Information

Since control-flow information is used mainly in the PDG building and not directly in our queries, we do not find new true positives or true negatives directly related to it. Nonetheless, several of the new true positives for *missing integrity check* described above involve data-flow indirection crossing files and are only detectable using a whole-project analysis.

We also find a number of false positives caused by a lack of control-flow information in all detectors. Specifically, for *empty password* smells, the main reason for high false positive rates is the existence of control-flow constructs that prevent the empty password from being used, either by skipping tasks or asserting that a variable is not empty. Listing 5.7 depicts an example in which

```yaml
- community.crypto.openssl_csr:
    privatekey_path: "{{ registry_dir_cert }}/domain.key"
  vars:
    registry_dir: /var/kubeinit/registry
    registry_dir_cert: "{{ registry_dir }}/certs"
```

Listing 5.6: Simplified example of a false *hardcoded secret* smell with data-flow indirection, adapted from `kubeinit/kubeinit`.

```yaml
# Contents of defaults/main.yml
bamboo_user: ''
bamboo_pass: ''

# Contents of tasks/main.yml
- name: create user
  user:
    name: "{{ bamboo_user }}"
    password: "{{ bamboo_pass }}"
  when: "bamboo_pass != ''"
```

Listing 5.7: Simplified example of a false *empty password* smell, adapted from `openmrs/openmrs-contrib-itsmresources`.

the variables initialised with empty values are meant to be overridden, and a conditional in the task prevents the user account from being created if the password is empty. All evaluated detectors report this as an empty password smell, but we consider this a false positive as the empty password can never be used in practice. Such false positives may be remedied in the future by a flow-sensitive analysis, which may be aided by the PDG representation.

Similarly, a lack of contextual information may lead to false positives of the *unrestricted IP address* smell. Listing 5.8 exemplifies a usage of the `0.0.0.0` IP address in a deny-rule in a firewall configuration that is falsely flagged as a security smell by all evaluated detectors. This rule instructs the firewall to block all traffic from any IP address by default, which is the opposite of the security weakness that the *unrestricted IP address* smell attempts to detect. However, all evaluated detectors ignore the type of firewall rule and simply report any usage of the IP address, causing false positives. Future work should investigate taking more contextual information into account to avoid such false positives.

String Patterns

Our improvements to the string patterns both uncovered new true positives and avoided other tools' false positives, e.g., in *missing integrity check* smells.

```yaml
- name: Create Extended acl deny_all
  arubaoss_acl_policy:
    acl_name: deny_all
    source_ip_address: 0.0.0.0
    destination_ip_address: 0.0.0.0
    protocol_type: PT_IP
    acl_action: AA_DENY
    acl_type: AT_EXTENDED_IPV4
```

Listing 5.8: Simplified example of a false *unrestricted IP address* smell, adapted from `aruba/aruba-ansible-modules`.

Although these improvements enabled GASEL to avoid many false positives in *hardcoded secret*, it also caused new false positives (not reported by other tools) for this smell. This again indicates a trade-off between precision and recall. Note that all three detectors suffer from low precision for *hardcoded secret*, possibly indicating that the string patterns are too general. Although refining the string patterns further may be a possible strategy to improve precision, we doubt that much can still be gained and question the maintainability effort required for such patterns. Future work could investigate whether an approach more akin to taint analysis, with literals as sources and specific parameters of a task module as vulnerable sinks, can improve these results. We believe that the data-flow information contained in the PDG representation could facilitate such an approach. Furthermore, instead of using manually-crafted string patterns, future work could train a machine learning model to predict whether a task argument name is security-sensitive (e.g., indicates a password).

For *unrestricted IP address*, we find that 5 of the false reports are caused by a bad string pattern which erroneously matched the IP `10.0.0.0`. Similarly, a number of false negatives for *admin by default* are because our string pattern requires a full match for the value, while GLITCH allows partial matches. Both regressions can easily be fixed in future work.

Composite Data

A major limitation of the PDG builder is that it considers composite data structures (lists and dictionaries) as mostly opaque data and does not always split their constituents into separate nodes. Therefore, GASEL lacks the ability to perform in-depth checking of this data. For instance, for *unrestricted IP address*, it misses many usages of bad IP addresses inside the composite data structures. Similarly, for *admin by default*, GLITCH can find a number of instances inside dictionary key-value pairs, which are not checked by GASEL. This limitation could be addressed by extending the PDG representation and smell queries further.

5.5.2 Files Ignored by GASEL

For RQ_1, we only focused on files that were scanned by GASEL. However, since GASEL relies on resolving dynamic inclusions in Ansible code, which is not always possible, it may miss files checked by other detectors. To investigate this limitation, we sampled 10 files per smell type that were checked by the other tools but not by GASEL, and investigated why GASEL ignored them.

We found that the main reason why GASEL fails to scan certain Ansible files is because of dynamic values that are difficult to approximate statically. GASEL ignores file inclusion if the file name depends on an expression, such as one that depends on the operating system name of the targetted machine. Although these files may contain security smells, GASEL cannot find them. Future work should investigate whether these dynamic file inclusions can be statically approximated so that their contents can be represented in the PDG.

Nonetheless, since GASEL only scans files it knows are reachable via a control-flow path from a playbook, it managed to avoid scanning a substantial number of irrelevant files. SLAC and GLITCH on the other hand, did scan these files, leading to many false positive reports. A majority of such files were test files, which developers often consider irrelevant [108]. Moreover, we observe that SLAC and GLITCH scanned Ansible files that are never included through an entry point, thus never executed and ignored by GASEL. Finally, we found a number of reports by the other tools in YAML files that do not contain Ansible code, such as Docker Compose and Kubernetes files, or even files containing plain data.

5.5.3 On the Importance of Control and Data Flow

Our investigation suggests that more than half of the security smells in Ansible are impacted by data-flow indirection (see RQ_3). However, this does not imply that detectors lacking data-flow information cannot detect such instances. Indeed, SLAC and GLITCH leverage variable naming to find potential secrets, and detection of unrestricted IP addresses or weak crypto algorithms does not require knowing where these values are used. Nonetheless, our evaluation (RQ_1) and consequent manual investigation (cf. Section 5.5.1) shows that taking data-flow information into account can lead to substantial improvements in both precision and recall, and we therefore recommend future research to follow this direction.

Although indirection through control flow is less prevalent (see RQ_3), we note that accounting for control flow is a necessity to accurately approximate data flow. Furthermore, several instances with indirect data flow require a whole-program analysis to detect. Moreover, we have shown that control-flow information can avoid scanning irrelevant files (cf. Section 5.5.2). Finally, control-flow information would be vital to address the low precision for *empty password* smells (cf. Section 5.5.1).

We note that *missing integrity check*, which was the least prevalent smell according to Saavedra and Ferreira [115], ranks fourth in our investigation.

Although our results are gathered from a different dataset and are thus not directly comparable, the substantially higher recall and precision obtained by GASEL still suggests that their approach has severely under-approximated the prevalence of this smell in practice. As shown earlier, data-flow information was the major reason for the improved recall for this smell, providing another motivation for its importance in security smell detection.

We also note that our approach detects much fewer *hardcoded secret* instances proportional to the total number of smells than prior research. This is likely in part because we do not consider usernames to be secret, per practitioner feedback [113]. However, we reiterate that our approach achieves the highest recall of all detectors for this smell, yet achieves low precision. This suggests that our results already over-approximate the number of hardcoded secrets, and prior research may over-approximate this even more.

5.5.4 Threats to Validity

We present our threats to validity according to the recommendations of Wohlin et al. [140].

A threat to *construct validity* stems from previously-discussed technical limitations and potential bugs in our PDG builder and smell detector which may cause false positives and negatives. To mitigate this threat, we conducted an extensive evaluation of our prototypes in addition to rigorous testing during its development.

The selection of the studied dataset may form a threat to *internal validity*. To mitigate, we applied well-established filtering criteria to maximise the quality of projects in this dataset. The construction of the oracle in RQ_1 exhibits some more threats to internal validity. First, we only considered files that were scanned by all three tools and may thus omit files that our approach missed. We partially mitigate this risk by qualitatively studying the missed files (Section 5.5.2). Second, we note again that the recall values reported in RQ_1 are an approximation, since our oracle is an under-approximation of the ground truth. Finally, the manual labelling of smell reports may introduce bias. However, this is mitigated by the labelling process being an objective binary decision following strict criteria, leaving little room for subjective influence.

As a threat to *external validity*, our findings cannot be generalised to other IaC languages such as Chef and Puppet. Nonetheless, we believe our approach to be sufficiently general to transpose to other languages.

5.6 Conclusion

The security of digital infrastructures and, by extension, the Infrastructure as Code scripts that configure them, is essential. However, existing approaches to detecting security smells in Infrastructure as Code disregard the analysed

scripts' control and data flow, and lack awareness of specific syntactic constructs. This causes them to miss numerous security smells, which may allow security weaknesses in IaC scripts to go unnoticed.

To address these limitations, this chapter presented an approach based on the Program Dependence Graph representation combined with Cypher graph queries to detect 7 security smells in Ansible code. The PDG provides vital data-flow information and can account for Ansible's syntactic particularities. Moreover, because its construction follows the control flow of Ansible scripts, our smell detection approach can disregard irrelevant and non-Ansible YAML files. We showed that these improvements enable our approach to outperform two state-of-the-art detectors on an oracle of 243 real-world security smells, with recall above 80% on 6 of the 7 considered security smells, while precision is above 90% for four smells.

We further investigated the prevalence of indirection caused by control and data flow in 7 933 security smells detected across 472 repositories and 3 457 Ansible entry points. Our findings show that over half of security smells involve data-flow indirection, and one in three smells involve dynamic file inclusion. A comparison of the frequency of security smells found by our approach to those reported in previous work suggests that previous work has severely underestimated the prevalence of security weaknesses. These findings strengthen the motivation to include control-flow and data-flow information into future security smell detection approaches.

Chapter 6

Software Composition Analysis for Ansible

Understanding *software supply chains*, i.e., the software components that make up a final product, is vital, especially from a security point of view. Attacks on software supply chains have become increasingly common [84] and their alarmingly high risk has led to new government legislations aimed at strengthening them, including in the United States[1] and the European Union[2]. Although this has attracted attention in the academic community, in Chapter 2, we found that the supply chain of infrastructure code remains an unstudied topic (cf. Section 2.3.3).

Because infrastructure code is used to deploy application code and its dependencies, we hypothesise that IaC's deployment supply chains may fundamentally differ from those for other software. In fact, deployment supply chains are more complex than those of application code, as they comprise third-party software from multiple independent software ecosystems. This includes not only third-party IaC artefacts within the code's own ecosystem, but also third-party software from other ecosystems. For instance, to provision a cloud machine, infrastructure code may need to depend on OS packages (e.g., to create cryptographic keys), the remote API of the cloud service (e.g., Amazon's AWS APIs), and development libraries (e.g., a language-specific wrapper around the remote API). This poses a challenge to practitioners, who may need several dependency management strategies for their deployment automation due to the different origins of its dependencies.

Ansible forms a particularly interesting case because of its reliance on *collections*. Recall from Section 2.2.5 that collections aggregate plugins that extend Ansible with new operations. A collection may depend on other collections, which can be specified in the collection's metadata manifest. Moreover, because a collection's plugins are implemented in a general-purpose programming language, usually Python, they can depend on development libraries from the ecosystem surrounding that language, such as Python libraries. Finally, they

[1] `https://www.nist.gov/itl/executive-order-14028-improving-nations-cybersecurity`

[2] `https://www.consilium.europa.eu/en/press/press-releases/2022/10/17/the-council-agrees-to-strengthen-the-security-of-ict-supply-chains/`

```
1  - hosts: web-servers
2    tasks:
3      - name: Install NodeJS
4        apt:
5          name: nodejs
6          state: present
7      - name: Deploy app from git repository
8        git:
9          repo: https://github.com/my/repo
10         dest: /app
```

Listing 6.1: Example of an Ansible playbook with plugin dependencies.

can also depend on other types of software, such as binaries installed through OS packages.

Listing 6.1 exemplifies a deployment playbook for an application. It installs the `nodejs` OS package through the `apt` package manager, and clones a git repository. Both are part of the application's run-time software supply chain. However, Ansible's `apt` module (line 4) depends on the `apt` Python package, whereas the `git` module (line 8) requires the `git` binary. As these are dependencies of the deployment code, they form the application's *deployment software supply chain*.

As illustrated, the deployment supply chain caused by Ansible playbooks comprises various types of third-party software from several sources. Although Ansible comes with a tool to manage dependencies on a collection, that tool does not manage the collection's dependencies on third-party software from other ecosystems. Furthermore, Ansible offers no mechanism to specify and coordinate these dependencies in a structured manner; they can only be specified informally in the plugin's documentation,[3] as exemplified in Figure 6.1. Moreover, as module plugins execute on remote hosts, their dependencies are to be installed on that remote host, rather than with the collection on the controller. Not installing these dependencies will cause the execution of the Ansible playbook using the plugins to fail partway, leaving the remote hosts in a partially configured state.

This lack of structured dependency specifications and automated dependency management may hamper the reliability and reproducibility of Ansible automation code. In fact, missing packages have been identified as the leading cause of crashes in automatically-generated Ansible playbooks [46]. Practitioners may also avoid adopting modules because of their dependencies,[4] instead opting to replace it with imperative commands, which are considered a bad practice [65]. Moreover, changes to a collection's dependencies can

[3] `https://github.com/ansible/ansible/issues/62733`
[4] `https://github.com/manala/ansible-roles/commit/134766b`

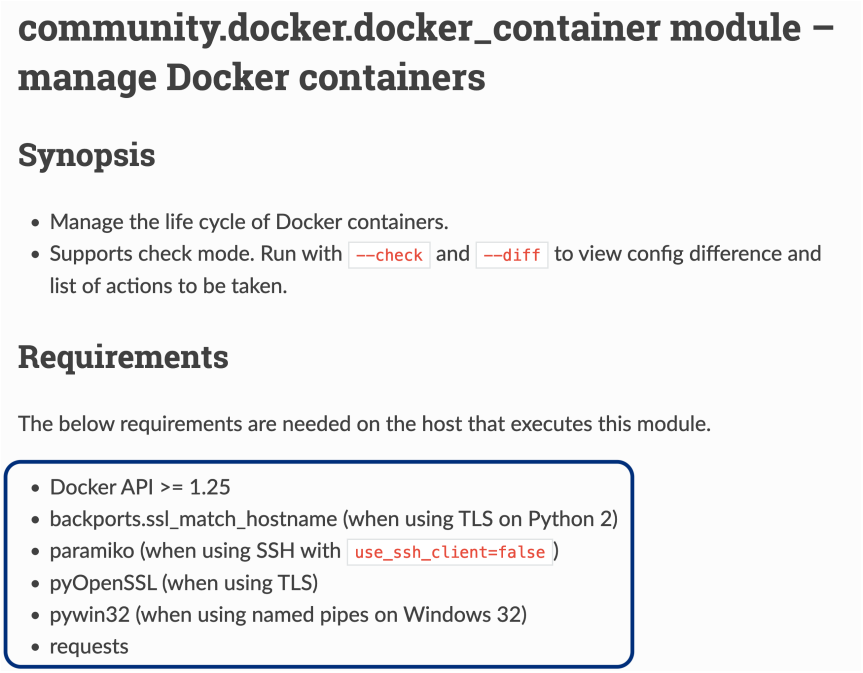

FIGURE 6.1: Example of an Ansible plugin's documentation, specifying requirements in unstructured semi-natural language.

cause problems for the collection's clients.[5] For instance, after an update to the *hetzner.hcloud* collection, its developers received several bug reports of the collection causing a crash.[6] The root cause was that the update requires a newer version of a Python dependency. Although this was noted in the collection's changelog, a lack of dependency management automation caused this to become a breaking change for numerous clients, eventually leading to the collection being pinned to an earlier version in the upcoming Ansible release.[7]

Without dependency management automation, Ansible effectively shifts responsibility to plugin developers and to plugin users. Moreover, without structured dependency specifications, identifying which third-party dependencies are needed and from where they originate becomes a challenge.

Therefore, in this chapter, we conduct an empirical study of deployment dependencies of Ansible Infrastructure as Code. Specifically, we manually study a large sample of documented requirements, from which we construct a taxonomy of the types of dependencies that participate in the deployment supply chains of Ansible plugins. Moreover, we qualitatively study the practices related to dependency management adopted by Ansible plugin developers. We focus on how they check whether dependencies are satisfied, and the steps taken when a dependency is not satisfied, from which we derive a catalogue of dependency management practices. As the unstructured dependency-specifying requirements do not lend themselves well to automated

[5] https://github.com/ansible/ansible/issues/35612
[6] https://github.com/ansible-collections/hetzner.hcloud/issues/211
[7] https://github.com/ansible-collections/hetzner.hcloud/issues/217

analysis, we design a Software Composition Analysis (SCA) that automatically identifies the dependencies of an Ansible plugin from its implementation. We implement and apply the SCA in a large-scale quantitative investigation of Ansible deployment supply chains.

The remainder of this chapter is structured as follows. Section 6.1 introduces the research questions investigated in this chapter and describes the design of our empirical study. Then, in Section 6.2, we present the results of our qualitative study, including the taxonomy of dependency types and dependency management patterns. Subsequently, in Section 6.3, we propose a Software Composition Analysis that employs the dependency management patterns to automatically identify dependencies from plugin implementations; we describe its evaluation, and we apply it in a large-scale quantitative analysis. Section 6.4 discusses the implications of our findings, providing recommendations to manage dependencies in Ansible plugins and conceptualising applications of our Software Composition Analysis. Finally, Section 6.5 describes related work, and Section 6.6 concludes.

A replication package containing the data, our analysis scripts, and the dependency extractor is available at `https://figshare.com/s/3e30dddab0fc7e00ee82`.

6.1 Empirical Study Design

In this section, we describe the design of our empirical study. We investigate the following research questions:

- *RQ_1: Which types of third-party software do Ansible plugins depend on?* We first perform a manual qualitative analysis of the types and characteristics of third-party software dependencies of Ansible plugins, to gain a better understanding of the types of software making up deployment supply chains.
- *RQ_2: How do Ansible plugin developers manage dependencies?* Since Ansible shifts dependency management to plugin developers, we qualitatively investigate whether and how plugin developers implement those responsibilities, to gain a better understanding of the state of the practice.
- *RQ_3: Can Ansible plugin dependencies be identified automatically?* As Ansible does not provide structured dependency specifications, we aim to design a Software Composition Analysis (SCA) that automatically identifies the diverse types of dependencies from Ansible plugin implementations.
- *RQ_4: How common are Ansible plugin dependencies?* Using the SCA resulting from RQ_3, we conduct a large-scale study of Ansible collection dependencies. We quantify the extent to which plugins depend on other software, through which we will gain a better understanding of how frequently plugin users may be faced with challenges caused by deployment supply chains.

6.1.1 Data Collection

To collect a representative dataset of Ansible collections, we scrape the Ansible Galaxy registry (cf. Section 2.2.6). For each collection, we use the registry's API to gather information such as its GitHub repository, its releases, and the number of times it has been downloaded from the registry. Then, we augment this data with information from the collection's GitHub repository (if specified), including the number of commits, issue count, star count, etc. We collected the data on January 26th, 2024, discovering 2 817 collections and 1 867 GitHub repositories. We could not collect the information of 175 of these GitHub repositories, e.g., because the repository was removed or set to private.

As our goal is to study unique, open-source Ansible collections, we exclude 1 125 of the 2 817 collections because they do not have a public GitHub repository. Furthermore, to avoid considering duplicates, when two collections specify the same GitHub repository, we only retain the one with the highest download count. This eliminates another 242 collections.

Second, we apply filtering to ensure that our dataset contains only mature and actively-maintained collections. We exclude 8 collections that are marked as deprecated on Ansible Galaxy. We further exclude another 584 whose repository has not been committed to in the year leading up to the data collection date, and are thus no longer actively maintained. To retain only those that are mature, we exclude 122 collections with fewer than 10 commits. Similarly, we exclude another 165 that have been active for less than half a year, measured as the time between the date of the first and last commit to the repository.

Finally, we want to focus on collections that are widely used in practice. Therefore, we retain only the top 10% most downloaded collections according to Ansible Galaxy statistics. This excludes another 384 collections that have been downloaded less than 15 700 times.

After filtering, the resulting dataset counts 187 widely-used, mature, open-source Ansible collections, representing a total of 92.3% of all downloads in the Ansible Galaxy ecosystem. Table 6.1 depicts their summary statistics. For the empirical analysis, we download the latest version of each collection. We also add the *ansible.builtin* collection to the dataset. It is shipped with Ansible and therefore should be included, but is missed by our data collection strategy since it is not distributed through Ansible Galaxy. Our final dataset contains 188 collections.

6.1.2 Parse Collection Documentation

We proceed to extract the requirements specified in the embedded documentation of Ansible collection plugins. The documentation for a plugin is embedded into its Python implementation as a YAML document. We create a script that, given a collection, first enumerates the collection's plugins, then extracts documented requirement lists for each plugin individually. For instance, for the plugin whose documentation is depicted in Figure 6.1, the

Table 6.1: Dataset statistics, excluding *ansible.builtin*.

Property	Minimum	Median	Mean	Maximum
# downloads	15.7K	136.3K	2.11M	62.5M
# commits	10	269	767	33.7K
Last activity	0 days	18 days	53 days	362 days
Active time	233 days	1 370 days	1 355 days	3 984 days

script would produce a list of 6 items, each containing the text for one bullet of the highlighted requirements list. For both steps, the script leverages Ansible's `ansible-doc` command, which processes the embedded documentation. We run the script on the latest version of the 188 collections in the dataset. We enumerate 13 721 plugins and successfully parse the documentation of 13 164 plugins. In total, we identify 10 960 requirements belonging to 5 537 plugins, forming the population for our study.

6.1.3 Open Coding of Collection Dependencies

We manually investigate a statistically significant sample of collection requirements to identify the types of third-party software on which plugins depend (RQ_1), as well as dependency management patterns in the plugin implementations (RQ_2). For the latter, we intend to focus on how plugins check whether dependencies are satisfied, and on what steps plugins take if a dependency is not satisfied.

We first deduplicate documented requirements across different plugins in the same collection, because these plugins may use common implementations, which would introduce bias in our results. This results in 866 unique combinations of a collection and an individual documented plugin requirement. We then take a random sample of this deduplicated population, using a sample size obtained from Cochran's formula for categorical data, adjusted for small population sizes [21] with 95% confidence, 5% margin of error, and an estimated population proportion of 50% to maximise the sample size. This results in a sample of 266 requirements spanning 51 collections. Then, we apply open coding to qualitatively answer RQ_1 and RQ_2. To avoid subjective bias which could be caused by the use of a single labeller, we only label purely objective properties for which any ambiguity can be resolved by consulting the online documentation.

For RQ_1, the labeller determines whether the documented requirement represents a valid software dependency (e.g., "This plugin requires the "requests" Python package"), as opposed to requirements that do not contribute to the supply chain and are thus outside the scope of this study (e.g., "This plugin requires root privileges"). Valid requirements state the type of dependency, such as the source ecosystem, and related properties, such as whether the dependency has a version constraint. When the dependency type is unclear,

TABLE 6.2: Example output of the open coding for RQ_1.

Requirement	Valid?	Type	Where?	Version?	Conditional?
`ncclient >= 0.5.3 when using netconf`	✓	Package: Python	Host	✓	✓
`cqlsh`	✓	Package: OS	Host	✗	✗
`scp if using protocol=scp with paramiko`	✓	Package: Python	Controller	✗	✓
`owner or maintainer rights to project on the GitLab server`	✗	—	—	—	—

the labeller refers to online documentation for the specified dependency. In subsequent iterations, they generalise the labels to create categories of dependencies, and derive the general properties exhibited by dependency-specifying documented requirements.

Table 6.2 illustrates this process using an excerpt of the results. For instance, in the first requirement, we identify `ncclient` as a valid dependency, and based on its online documentation, we determine it to be a Python package. We also identify that the requirement contains a version constraint ("`>= 0.5.3`") and is only needed when a certain condition is met ("`when using netconf`"). Finally, based on the type of the plugin, we have determined the requirement to be needed on the Ansible hosts rather than the controller. Similarly, for the second requirement, we identify `cqlsh` as a binary executable that is installed from an Operating System package, and for the third requirement, we identify `scp` as a Python package required on the Ansible controller. Finally, we determine that the fourth requirement does not specify a valid software dependency, and thus mark it as invalid for our purposes.

For RQ_2, for each dependency, we randomly select one plugin whose requirement list contains the dependency. The labeller then inspects the entirety of this plugin's implementation for dependency management concerns. They summarise the means through which the plugin checks whether the dependency is satisfied, and the behaviour the plugin exhibits in case the dependency is not satisfied. Both summaries are assigned as distinct codes for the dependency. After coding each dependency, the labeller generalises the codes to a set of high-level properties that characterise the dependency management, and iteratively refines the properties to capture variations in similar codes. The result of this process is a set of commonly-occurring implementation patterns related to collection dependency management.

To exemplify, Listing 6.2 depicts a dependency management implementation for a dependency on the `selinux` Python package. During labelling, we note several aspects in this implementation. First, we identify that the implementation wraps the import statement for this dependency in a try-except

```
SELINUX_IMP_ERR = None
try:
    import selinux
    HAS_SELINUX = True
except ImportError:
    SELINUX_IMP_ERR = traceback.format_exc()
    HAS_SELINUX = False

...

def main():
    ...
    if not HAS_SELINUX:
        module.fail_json(
          msg=missing_required_lib('libselinux-python'),
          exception=SELINUX_IMP_ERR)
    ...
```

Listing 6.2: Example of a dependency management implementation in the `selinux` module of the *ansible.posix* collection.

block, and sets a flag and an exception variable in case the import fails. We also note that this flag is checked in the plugin's `main` function. This constitutes the plugin's mechanism to check whether the dependency is satisfied. Then, we turn to the behaviour it exhibits when the dependency is not satisfied, and determine that the plugin fails execution through the `module.fail_json` function. We also note that it uses the `missing_required_lib` function and the exception variable to construct a failure message. We construct such summaries for all checked implementations, and afterwards generalise them to abstract common aspects into dependency management patterns.

6.1.4 Automated Software Composition Analysis

After the qualitative analysis in RQ_2, we will have identified code patterns whose presence in a plugin's implementation are indicative of its management of a dependency. As unstructured dependency-specifying requirements cannot be used to reliably extract deployment supply chains, we now use these patterns to design a Software Composition Analysis that automatically extracts managed dependencies from Ansible plugin implementations.

To find pattern instances in a plugin implementation, we will use Joern [61], a multi-language static analysis framework. Joern represents projects as code property graphs (CPGs), the nodes of which represent program elements such as statements and expressions. The edges capture properties of and relationships between these elements, including abstract syntax tree containment,

calls between functions, control flow, data flow, and type information. Our approach will build a CPG for all plugins in an Ansible collection, and will evaluate Joern queries against the CPG that find instances of dependency management patterns and thereby identify dependencies (cf. Section 6.3).

To answer RQ_3, we will measure the precision and recall of our Software Composition Analysis. For recall, the ground truth comprises the manually-validated dependency-specifying documented requirements from RQ_1. We run the SCA on the implementation of the plugins from the ground truth, and manually compare its output to the ground truth to identify true positives. When the SCA reports multiple results that belong to the same dependency (e.g., individual binaries from one OS package), we manually merge the reports into one, consulting online documentation where necessary. We do not distinguish between optional and mandatory dependencies, and do not consider version constraints, as we only need to determine whether and which third-party software plugins rely on.

To compute recall, we mark those entries from the ground truth that are missing from the SCA output as false negatives. Note that when a plugin implementation does not check for the presence of its dependencies, the SCA will fail to identify them. Therefore, we calculate an upper and lower bound on recall. The upper bound is defined as the ratio of identified dependencies over the number of dependencies in the ground truth for which the plugin checks whether the dependency is satisfied. The lower bound is the ratio of identified dependencies over the entire size of the ground truth, including unchecked dependencies.

To calculate precision, when an entry in the SCA output is not in the ground truth, we further inspect the plugin documentation and implementation to determine whether it is a true or false positive. This is necessary because some plugin dependencies may not have been documented, and are therefore missing from the ground truth which is constructed from the plugin documentation. Moreover, some dependencies may be on built-ins, such as built-in Python libraries or OS packages. Although such built-in dependencies are likely to be considered as false positives by plugin users and maintainers, they are depended upon by the plugin. Moreover, dependencies that are typically considered built-in may turn out to be unavailable after all, e.g., the Python `ssl` library may be missing if Python is compiled without SSL support.

Therefore, whether built-in dependencies should be considered true positives depends on context. To avoid subjectivity, we again calculate an upper and lower bound on precision. For the lower bound, we assume that *all* built-in dependencies are false positives, which would be appropriate for developer-oriented tooling. For the upper bound, we do not distinguish between built-in and other dependencies, which would be appropriate to extract the entire supply chain.

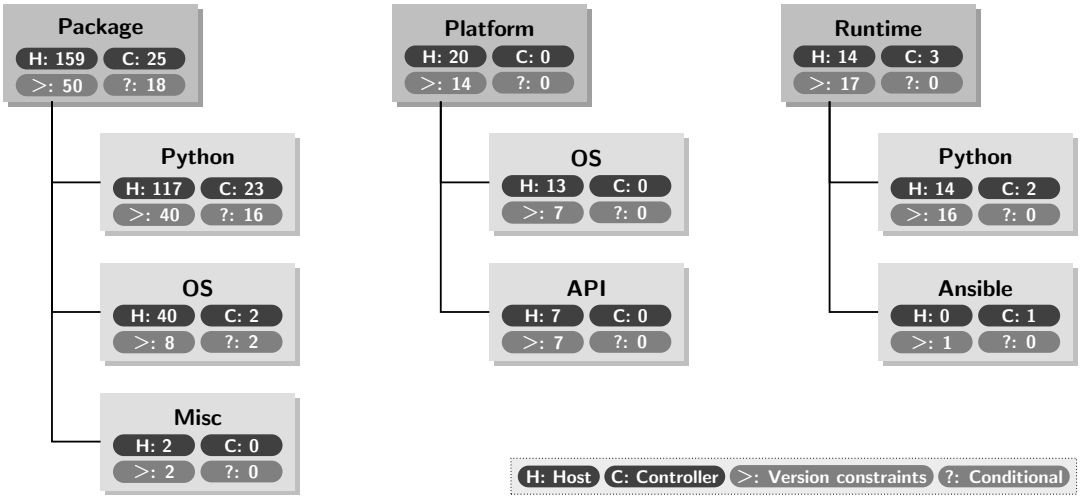

FIGURE 6.2: Taxonomy of dependencies in Ansible collections. Host and controller counts are mutually exclusive.

6.1.5 Quantitative Analysis of Collection Dependencies

To answer RQ_4, we apply the SCA resulting from RQ_3 to all 188 collections in the dataset, obtaining lists of dependencies for each plugin. We map each code pattern identified in RQ_2 to the most common dependency type the pattern is used for. We use the mappings to automatically identify the types of third-party software in the SCA results. Using the results, we investigate and compare how many plugins depend on the types of third-party software identified in RQ_1.

6.2 Qualitative Analysis

In this section, we present the results of our manual qualitative study into the dependencies of and the dependency management implemented by Ansible plugins.

6.2.1 RQ_1: Which types of third-party software do Ansible plugins depend on?

Among the sample of 266 requirements from the embedded plugin documentation, we manually identify 221 valid dependency-specifying ones. The other 45 requirements specify generic preconditions, such as user permissions or conditions on the Ansible configuration. We note that the dependency specifications in the plugin documentation are highly unstructured, ranging from merely the dependency name (e.g., "requests") to full sentences (e.g., "Requires `lzma` (standard library of Python 3) or `backports.lzma` (Python 2) if using `xz` format").

In the valid dependencies, we discern 7 distinct types, grouped into 3 categories. The resulting taxonomy is depicted in Figure 6.2. The most common category is *Package*, comprising third-party software that is typically

installed through managers. Python libraries form the majority of this category, followed by operating system packages for package managers like `dpkg` (Debian, Ubuntu, . . .) or `rpm` (CentOS, Fedora, . . .). The *Misc* subcategory encompasses other package dependencies that are less common, such as Terraform packages. The second category is *Platform*, consisting of operating systems (e.g., macOS) and APIs. The latter refers to interfaces the plugins communicate with, either locally (e.g., the Docker API) or remotely (e.g., remote management systems). The final category consists of *runtime* version dependencies, encompassing minimum versions of Ansible and Python.

We identify 3 orthogonal properties of the 7 categories of collection dependencies. First, we can classify dependencies according to whether they should be installed on the host that is being configured or on the Ansible controller (cf. Section 2.2), which depends on the type of plugin. 86% (190) of the studied dependencies must be satisfied on the remote host machines. We note that host dependencies are more diverse than controller dependencies. Indeed, most controller dependencies are Python packages, while certain types of dependencies, such as those in the *Platform* category, are only needed on the remote host. Second, only 37% of dependency specifications (81) include some form of version constraint, with API and runtime dependencies always specifying a version constraint. These are denoted by `>` in the taxonomy. Finally, while the majority of dependencies are mandatory, we find 18 (8%) that are either optional or only need to be installed depending on certain conditions, e.g., depending on the arguments given to the plugin. These are denoted by `?` in the taxonomy.

Answer to RQ_1: Plugin documentation does not always specify dependencies, nor in a structured manner. We discern 7 types of Ansible plugin dependencies in 3 categories. Python libraries and OS packages are the most prevalent. Most requirements are needed on hosts. Version specifications are uncommon.

6.2.2 RQ_2: How do Ansible plugin developers manage dependencies?

We find that for the majority of dependencies (174 of 221), the plugins check whether the dependency is satisfied. The majority of these checks (67%) happen inside the plugin file itself, while the remaining 33% occur in shared utility files that can be used by multiple plugins which may have different dependency or version requirements.

However, for 47 dependencies (21%), the plugin does not check that it is satisfied, and only 36% of specified version constraints are verified by the plugins. In the *Runtime* category, only 2 Python version requirements are validated by plugins, with the remaining 14 Python versions and 1 Ansible version remaining unchecked. Omitting these checks may lead to run-time crashes with confusing error messages.

TABLE 6.3: Catalogue of dependency management implementation patterns.

Pattern	Use cases	Example	Detection pseudo-query

Guarded import	Python packages, Python version	`HAS_LIB = True`
		`try:`
		`  import lib`
		`except ImportError:`
		`  HAS_LIB = False`
		`if not HAS_LIB:`
		`  <error>`
			`tryStatement`
			`  .where(`
			`  _.exceptBlock`
			`  .contains(_.isAssignment))`
			`  .tryBlock.children`
			`  .filter(_.isImport)`
			`  .map(_.importedName)`

Dynamic import	Python packages	`try:`
		`  importlib`
		`  .import_module("lib")`
		`except ModuleNotFoundError:`
		`  <error>`
			`callTo("import_module")`
			`  .arguments(0)`
			`  .resolveString()`

community.general `deps`	Python packages	`with deps.declare("lib"):`
		`  import lib`
		
		`deps.validate(module)`
			`withStatement`
			`  .where(`
			`  _.expression`
			`  .isCallTo("deps.declare"))`
			`  .body.filter(_.isImport)`
			`  .map(_.importedName)`

`get_bin_path`	OS packages, OS, Python packages	`bin = module.get_bin_path(`
		`  "binary_name")`
			`callTo("get_bin_path")`
			`  .arguments(0)`
			`  .resolveString()`

community.general `CmdRunner`	OS packages	`runner = CmdRunner(`
		`  module, "binary_name")`
			`callTo("CmdRunner")`
			`  .arguments(1)`
			`  .resolveString()`

Dependency Management Patterns

We identify 5 common implementation patterns for checking whether dependencies are satisfied. Table 6.3 summarises them with an example, whereas Figure 6.3 depicts their frequency per dependency type.

The most common pattern is the *guarded import*, in which developers wrap an import of a Python package in a try-except block. They then assign a variable to indicate whether the import was successful, and check this variable to abort with an error message. Its frequent usage for Python packages is not surprising, as it is recommended in the Ansible documentation[8]. *Dynamic import* is a closely-related but less common pattern in which developers import a Python package dynamically with the `importlib` library that is built into Python.

A common pattern for OS packages is to use the `get_bin_path` function provided by Ansible, which takes the name of a system binary and resolves it to an absolute path. If the binary cannot be found, it either returns an empty value or raises an exception, depending on its arguments. These results can be inspected to check whether a binary exists on the system, and is thus often used to check for OS packages and platforms.

Finally, we find two patterns specific to the *community.general* collection, a prominent collection in our dataset. This collection offers two utilities to

[8] `https://docs.ansible.com/ansible/latest/dev_guide/testing/sanity/import.html`

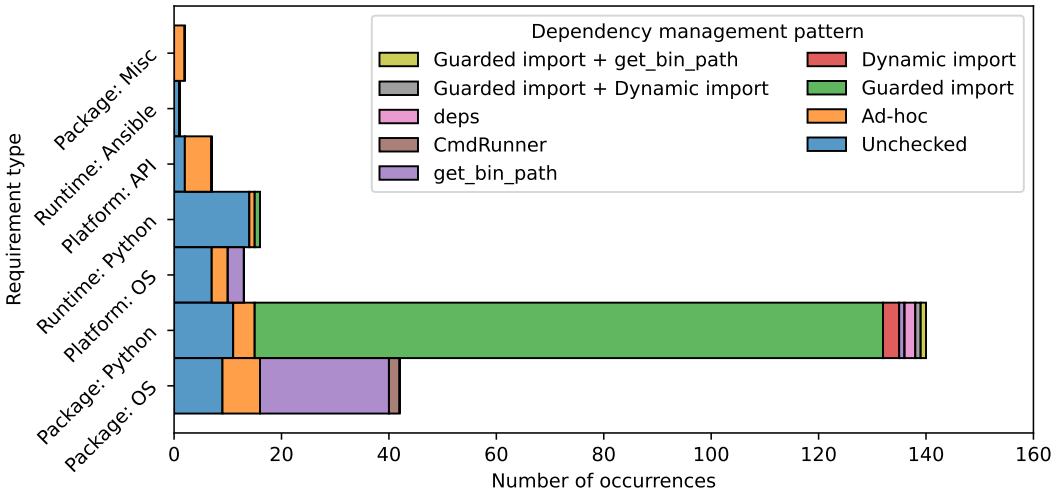

FIGURE 6.3: Frequency of dependency management patterns.

interact with dependencies. The `deps` utility offers an abstraction on top of the *guarded import* pattern using a context manager, whereas the `CmdRunner` class is an abstraction to interact with system binaries which internally calls `get_bin_path`. Since their usage may be widespread across the *community.general* collection, we include them in the catalogue.

We do not find any implementation patterns for runtime versions, API platforms, or miscellaneous packages as these are either not checked, or rely on ad hoc implementations. We also note that in rare cases, a plugin may use multiple mechanisms to check the same dependency.

Failure Patterns

Of the 174 plugin dependencies that are checked, in 166 cases the plugin simply fails if the dependency is not satisfied. We only found 5 cases in which the plugin automatically installs the dependency, and 3 cases in which the plugin proceeds with degraded functionality.

We recognise 3 patterns in the failure behaviour of plugins. For 105 dependencies, the plugin aborts through Ansible's `fail_json` function, which fails the module's execution and returns structured output describing the failure reason. In another 26 dependencies, the plugin raises its own exception, whereas in the last 35 cases, the plugin propagates an exception from one of the implementation patterns (e.g., an exception raised by a `get_bin_path` call).

Orthogonally, to create an error message, the plugin can use Ansible's `missing_required_lib` function, which takes the name of a dependency and constructs an error message explaining the missing dependency and how it can be installed. However, we found it is only used in a third of all failure behaviours. Specifically, we observe such a call with 54 instances of the `fail_json` pattern, and with 3 instances of the exception raising pattern.

Answer to RQ_2: While 79% of Ansible plugins check their dependencies, only 5 automatically install them. Only 36% of specified version constraints are verified. Checks for Python and OS packages often follow a discernable pattern.

6.3 Automated Software Composition Analysis

RQ_1 showed that dependency types are diverse and that the documentation that specifies dependencies is informal and unstructured, hindering the ability to automatically extract dependencies from documented requirements. Having identified 5 dependency management patterns in RQ_2, we investigate whether the dependencies of a plugin can be extracted automatically from its implementation instead (RQ_3). To this end, we create a Software Composition Analysis (SCA) for Ansible plugins based on the Joern framework [61]. The analysis operates in two phases. First, it matches the patterns against individual functions in a collection's source code. However, these patterns may occur in common utility functions instead of the plugin implementation itself (cf. RQ_2). Therefore, in the second phase, the SCA propagates the identified dependencies to each function's transitive callers, thereby propagating the dependencies from common implementations to each plugin that uses them.

6.3.1 Semantic Matching of Dependency Management Patterns

We implement CPG queries for each of the 5 patterns, which the last column of Table 6.3 summarises as Scala-like pseudocode. As the concrete implementation of a pattern can vary in source code, the queries rely on semantic information such as data-flow information to overcome the limitations of purely syntactic pattern matching.

The query for *guarded imports* finds try-except blocks that contain an import, and extracts the variables assigned in the block. It then uses data-flow information to find usages of those variables inside of conditions (e.g., in an if-statement), indicating conditional execution based on the result of the import. Finally, it marks the function containing the condition with the dependency names extracted from the import.

The query for the *community.general* `deps` pattern identifies `with` statements using the `deps.declare` function. It marks the functions containing a call to `deps.validate` with the dependency names extracted from the import statements in the with-block.

The queries for the *dynamic import*, `get_bin_path`, and *community.general* `CmdRunner` patterns all search for calls to the respective function. Functions containing such calls are marked with the dependency name obtained from the call's arguments. When the argument is a string literal, the name can be extracted straightforwardly. However, the argument can also be a variable

reference, in which case the query attempts to resolve this reference to a single constant definition of a string literal. In case the reference cannot be resolved, resolves to multiple possible definitions, or is not a string literal, we cannot confidently extract the dependency name and therefore under-approximate by omitting the call.

6.3.2 Match Propagation

Because multiple plugins may use the same utility functions (cf. RQ_2), dependencies in such functions need to be propagated to the plugins that use them. Therefore, we transitively propagate the dependencies backwards along the call graph to the function's callers. Due to Python being dynamically typed, the call graph constructed by Joern is an approximation. Specifically, if Joern cannot infer type information for the receiver of a method call, it resolves calls by name. This can lead to a vast over-approximation for generic method names like "run", which risks generating many false positives. Therefore, our propagation mechanism is conservative, and only propagates dependencies if the callee's receiver type is known, or if the function name is unique in the project.

We also noticed several shortcomings in Joern's call graph construction for object-oriented Python, such as missing call graph edges for super calls and self calls. As these limitations hamper our dependency propagation, we implement additional post-processing passes on the CPG to add the missing edges.

6.3.3 RQ_3: Can Ansible plugin dependencies be identified automatically?

The ground truth, constructed from the manually-investigated sample of valid dependency-specifying documented requirements of RQ_2, comprises 221 dependencies spread across 166 plugins. Running the Software Composition Analysis for these plugins results in 300 dependencies for 127 plugins. 57 of the results reported by the SCA are individual parts of the same dependency, e.g., individual Python packages such as `win32pipe` and `win32event`, both belonging to the `pywin32` Python dependency. We aggregate these 57 results into 17 groups, as described in Section 6.1.4, leading to 260 unique extracted dependencies.

Recall

The SCA correctly identified 135 dependencies out of the 221 unique dependencies in the entire ground truth, leading to a lower bound on recall of 61.09%. However, considering only those dependencies for which a check is implemented, the ground truth size decreases to 176, leading to an upper bound on recall of 76.7%. Table 6.4 depicts lower and upper bounds for individual dependency types. Python packages, the most common dependency type, can be identified with high recall, while around half of the OS packages

Table 6.4: Recall per dependency type.

Dependency type	Lower bound		Upper bound	
Package: Python	79.3%	111/140	87.4%	111/127
Package: OS	45.2%	19/42	57.6%	19/33
Package: Misc	0.00%	0/2	0.00%	0/2
Platform: OS	30.8%	4/13	57.1%	4/7
Platform: API	0.00%	0/7	0.00%	0/5
Runtime: Python	6.25%	1/12	50.0%	1/2
Runtime: Ansible	0.00%	0/1	*N/A*	0/0

```
1 IPTABLES = {'ipv4': 'iptables', 'ipv6': 'ip6tables'}
2 module.get_bin_path(IPTABLES[ip_version], True)
```

Listing 6.3: A missed dependency due to complex data flow.

can be identified. Other dependency types are often managed ad hoc and thus cannot be detected with high recall.

We investigated the root causes of false negatives in more detail. In 13 cases, the false negatives were caused by the SCA failing to match a dependency pattern because the dependency was checked with an ad hoc implementation. For another 8 false negatives, the SCA failed to match their code against a pattern because the data flow was too complex. Listing 6.3 provides an example of an implementation that was not matched because the argument to `get_bin_path` could not be resolved to a single literal definition. In another 12 cases, the SCA matched the implementation with a pattern but failed to propagate the dependency through the call graph due to a lack of type information on the receiver objects of method calls. Further improvements to Joern's type inference could likely alleviate this issue without requiring changes to our approach.

Precision

Across the 260 grouped results, we find 193 non-built-in true positives, and 27 built-in true positives. This corresponds to a precision between 74.2% and 84.6%. We also find 30 dependencies (13.7%) that are managed by the plugin but not mentioned in the documentation.

We again identify root causes for false positives. The most common root cause (23 false positives) occurs only in the *ansible.builtin* collection. As its implementation is part of Ansible itself, the SCA inspects the entire Ansible implementation. This caused it to extract Ansible's own dependencies, which it then erroneously propagated to the built-in plugins, although they are not

dependencies of the plugins themselves. In practical applications, omitting *ansible.builtin* would raise precision to between 89.5% and 95.0%.

Answer to RQ_3: Dependency management patterns can be used to automatically extract Ansible plugin dependencies, with recall between 61% and 77%, and precision between 74% and 95%. The dependency SCA complements the plugin documentation by identifying undocumented dependencies.

6.3.4 RQ_4: How common are Ansible plugin dependencies?

To investigate the dependencies of Ansible collections at scale, we applied the Software Composition Analysis on 11 241 plugins from 187 of the 188 collections in the dataset. We exclude plugins from *ansible.builtin* as they caused too many false positives in RQ_3 due to the SCA reporting Ansible's own dependencies. To map dependency management patterns to dependency types, we consider usages of *guarded imports*, *dynamic imports*, and *community.general* `deps` to signify Python packages, whereas `get_bin_path` and *community.general* `CmdRunner` signify OS packages.

Across the 11 241 plugins, we find no dependency management pattern in 62% of plugins, meaning many plugins might not use any dependency. Another 35.9% depend on Python libraries, whereas 1.8% depend on OS packages. A mere 30 plugins (0.2%) depend on both. The large difference in the number of Python library and OS package dependency management patterns is consistent with our qualitative findings in RQ_1, especially considering that around 1 in 3 OS package dependencies are either not checked, or checked ad hoc. These results show that most plugins do not have (identifiable) dependencies, and when they do, they depend only on one type of third-party software.

However, when aggregating all plugins in a collection, we find that 51% of the collections have at least one dependency. 40.6% of the collections depend on Python libraries, 6 collections (3.2%) depend on OS packages, and 14 collections (7.5%) depend on both. Users of Ansible code that depends on a collection thus transitively depend on the collection's dependencies, creating extensive deployment supply chains sourced from multiple software ecosystems.

Figure 6.4 depicts the distribution of the number of unique Python library and OS package dependencies for plugins and collections. Each letter-value plot considers only plugins or collections with at least one dependency of the given type. We observe that the majority of plugins and collections depend on multiple Python libraries, sometimes as many as 10 individual libraries in a single plugin. Plugins generally depend on fewer OS packages, yet of the 20 collections that depend on OS packages, 19 depend on multiple. However, the SCA may report multiple binaries of the same OS package separately (cf. RQ_3)

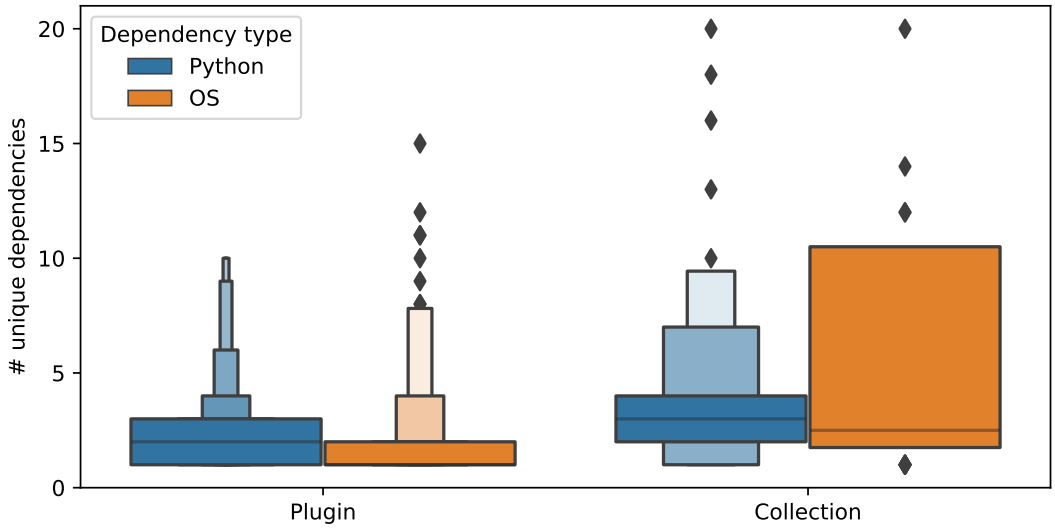

FIGURE 6.4: Letter-value plots depicting the distributions of the number of unique extracted dependencies for plugins and collections with at least one dependency pattern for a given type. An extreme outlier caused by the *community.general* collection is omitted.

which we cannot group automatically, so the number of OS packages required is likely lower in practice.

Table 6.5 lists the 10 most downloaded collections, the number of unique dependencies in the collection, and descriptive statistics on the number of dependencies of their plugins. We observe an extreme outlier (hidden in Figure 6.4), namely the *community.general* collection, which depends on 126 Python packages and 205 OS binaries. This collection was created to house many miscellaneous, unrelated plugins that used to be part of the Ansible core codebase, but are now all installed together through the same collection. Many of these plugins have their own unique dependencies, as suggested by the low number of mean and median dependencies per plugin. Other collections, such as *amazon.aws* and *community.docker*, instead rely on few dependencies, yet most of their plugins use several of them.

Finally, Table 6.6 lists the Python and OS packages that occur most frequently as collection dependencies. In terms of Python packages (Table 6.6a), we observe that most dependencies are utility libraries, such as `requests`, which facilitates network requests, `lxml`, which parses XML, and `cryptography`, which performs cryptographic operations. We also identify other types of commonly-checked packages, such as `ansible` and `ansible_collections`, indicating that collections import utilities from either Ansible or other collections. For OS packages (Table 6.6b), we observe a range of different tools, such as development binaries (e.g., `git` and `make`), OS commands (e.g., `mount` and `umount`), and package managers (e.g., `rpm` and `pkg`). However, we note that even the most popular binaries are depended on by few different collections.

TABLE 6.5: Dependency statistics for the 10 most downloaded collections.

			# unique packages		# dependencies per plugin	
Collection	**# downloads**	**# plugins**	**Python**	**OS**	**Mean**	**Median**
community.general	62.54M	724	126	205	0.83	1.00
ansible.posix	32.54M	24	4	12	0.71	0.00
amazon.aws	29.16M	118	5	0	2.88	3.00
ansible.netcommon	22.82M	26	16	0	0.88	1.00
ansible.utils	22.56M	65	8	0	1.55	2.00
community.docker	20.97M	41	13	2	5.12	4.00
ansible.windows	18.49M	44	0	0	0.00	0.00
kubernetes.core	15.95M	24	3	1	0.71	0.00
awx.awx	14.80M	48	4	0	1.06	1.00
community.crypto	14.49M	40	5	8	1.61	1.00

TABLE 6.6: Top 10 most common dependencies.

(A) Most common Python packages.

Package	**# coll.**	**# plugins**
`requests`	22	865
`ipaddress`	14	59
`yaml`	13	71
`ansible_collections`	10	942
`cryptography`	10	318
`lxml`	9	65
`ansible`	8	278
`xmltodict`	8	63
`paramiko`	7	29
`packaging`	6	470

(B) Most common OS packages.

Package	**# coll.**	**# plugins**
`openssl`	3	10
`git`	3	5
`rpm`	3	3
`make`	3	2
`mount`	2	5
`terraform`	2	4
`umount`	2	4
`gpg`	2	3
`pkg`	2	3
`ssh`	2	3

Answer to RQ_4: 38% of Ansible plugins and 51% of Ansible collections depend on software originating from other software ecosystems. Plugins often depend on multiple Python libraries, yet rarely on multiple OS packages.

6.4 Discussion

In this section, we discuss the implications of our findings.

6.4.1 Implications for Ansible Plugin Users

The results of RQ_1 show that the supply chain of an Ansible playbook may consist of software originating from multiple ecosystems. Within the Ansible ecosystem, practitioners may depend on Ansible itself, third-party Ansible code (e.g., Ansible roles), and Ansible collections. Our study has shown that they may also depend on Python libraries, OS packages, platforms, and runtime versions via a collection's plugins. In RQ_4, we uncovered that nearly 40% of plugins and over half of collections have at least one such dependency. Moreover, RQ_2 showed that plugins rarely install their dependencies themselves, and often fail to thoroughly check whether dependencies and their version constraints are satisfied. Although the dependencies of individual plugins mostly originate from single ecosystems (cf. RQ_4), deployment code can depend on many plugins, expanding the deployment supply chain.

This may complicate managing deployment supply chains of Ansible projects. Practitioners not only need to ensure that these dependencies are satisfied, they also need to ensure that these dependencies are up-to-date with patches for bugs and security vulnerabilities, while maintaining compatibility with the collections that require the dependencies. For general-purpose programming languages, this is facilitated by dependency management tools, such as Python's pip, Java's Maven, or JavaScript's NPM. In contrast, although Ansible's dependency management tool manages dependencies within the Ansible ecosystem, it does not manage the plugins' dependencies on software from other ecosystems. This is in part caused by the diversity of plugin dependencies, as Ansible cannot make assumptions about how such dependencies should be managed.[9] However, **although IaC can create reproducible environments for deployed applications, without automatic management of these plugin dependencies, IaC's own deployment environment may become unreproducible**. Therefore, future work could investigate how Ansible end users manage plugin dependencies, and devise new techniques to perform automatic dependency management across different ecosystems.

[9] `https://github.com/ansible/ansible/issues/62733`

6.4.2 Implications for Ansible Plugin Maintainers

RQ_1 showed that most plugins do not specify version constraints, and RQ_2 showed that even when they do, they are rarely checked. Moreover, RQ_2 showed that almost no plugins install their own dependencies. Combined with a lack of dependency management tools, this substantially hinders the reproducibility of IaC scripts. Therefore, **we recommend Ansible plugin maintainers to use structured, consistent, and machine-readable dependency specifications, and to adopt dependency management patterns.**

For IaC controllers, containerisation can be used to create reproducible environments, such as Ansible's *execution environments*[10]. However, for remote hosts, other IaC dependency management solutions are required. Of the host dependency types (cf. RQ_1), Python packages are the most common. We noticed that these are often implementation details, such as utility libraries used to make HTTP requests or wrappers around a remote API. For such dependencies, we argue that the IaC user should not need to be aware of the dependency, and that the IaC framework should take care of its installation. Conversely, the other dependency types are typically inherent to the plugin's functionality, e.g., a plugin that manages Docker containers requiring Docker to be installed on the remote host. We believe that automated installation is less pressing in such cases, as it would be nonsensical to use the plugin in case the dependency is not already satisfied.

To automatically manage Python library dependencies, maintainers could clone the dependency's source code into the collection's utilities, which get automatically installed with the collection. This "clone-and-own" strategy was adopted by the *hetzner.hcloud* collection[11] after an incident involving mismatched versions in its dependencies. This effectively prevents version conflicts and reduces the complexity for users, who no longer need to install the library themselves. However, it introduces additional complexity for maintainers due to the problems caused by "clone-and-own" reuse, such as bug and vulnerability propagation [58, 77, 118]. Additional support within Ansible itself to specify and install dependencies may be required to alleviate the complexity further. For instance, the Ansible runtime could install a compatible version of a dependency before running the plugin, and optionally remove this dependency after execution is finished.

In the absence of Ansible-provided support for specifying the dependencies, the taxonomy constructed in RQ_1 and dependency management patterns identified in RQ_2 may prove useful for maintainers as a reference on how to implement dependency management in their plugins. Nonetheless, we note that for several of the dependency types in our taxonomy, we could not find any implementation patterns. For certain types, such as remote APIs, this is to be expected, as every API and therefore every check will be different. However, for other types, notably Python runtime versions, we find only few checks and no patterns, although one may expect many of these checks to be similar.

[10] `https://docs.ansible.com/ansible/latest/getting_started_ee/index.html`
[11] `https://github.com/ansible-collections/hetzner.hcloud/pull/244`

This suggests that certain types of dependencies may often be overlooked by developers, which may cause version incompatibilities or unreproducible deployments.

6.4.3 Applications of the Software Composition Analysis

The Software Composition Analysis created and evaluated in RQ_3 could form the basis for practical tooling for Ansible developers. For instance, the SCA could be used to automatically report which dependencies are necessary to run an Ansible script. For plugin developers, the technique could be leveraged to identify undocumented dependencies or missing dependency management patterns. Finally, the SCA may also be useful as part of a toolchain to generate a Software Bill of Materials (SBOM) for cloud deployments. The SBOM could list the software components used to deploy an application, such as the Ansible collections, plugins, and their dependencies. Moreover, an SBOM could be distributed alongside every Ansible collection to document its components. This would enable practitioners to easily check the components of their deployment pipeline, e.g., to uncover security vulnerabilities, outdated dependencies, licence violations, etc.

The SCA could also be applied to study the evolution of Ansible collection dependencies. We conducted a preliminary study suggesting that nearly 40% of plugins and 86% of collections with dependency management patterns have undergone changes to the patterns. However, we observed little to no co-evolution with the documented requirements. A manual investigation of a sample of these changes also revealed that changes to the patterns are often refactorings, such as developers introducing a dependency management pattern for a previously-unchecked dependency. This preliminary experiment suggests that dependencies may evolve over time, which may bring about challenges for end users, but that the documentation may also be outdated or incorrect. Future work could study dependency changes in more depth.

6.4.4 Limitations of the Software Composition Analysis

In RQ_3, we found that our extractor can identify Python library dependencies accurately. However, it struggles to extract other dependency types and unchecked dependencies.

For unchecked Python libraries, one strategy may be to consider *all* import statements, including unguarded ones. However, this may extract many trivial built-in packages provided either by Ansible or Python itself, causing high false positive rates. Moreover, one would need to discern between imports of first-party and third-party code.

For OS packages, the extractor identifies implementation patterns used to check for the presence of a binary, causing it to miss unchecked OS dependencies whose binaries are used directly. Therefore, it may be possible to identify unchecked dependencies by also matching such direct usages.

We found no implementation patterns for Python runtime versions (cf. RQ_2) and thus fail to identify many of those dependencies. Future work could investigate whether other techniques could identify required Python versions, e.g., inspecting syntax features or imported built-in packages [153].

Finally, the `missing_required_lib` function (cf. RQ_2) may be useful to identify dependencies that are managed through ad hoc implementations. However, we note that it is used in less than half of the dependency management implementations, and that its presence does not convey the dependency types.

6.4.5 Threats to Validity

We present our threats to validity according to the recommendations of Wohlin et al. [140].

As a threat to *internal* validity, the manual investigation performed in RQ_1 and RQ_2 was performed by a single labeller, which may lead to subjective bias. We mitigated this threat by limiting ourselves to purely objective observations and consulting online documentation in case of ambiguity. Moreover, we avoid subjectivity in RQ_3 by providing a lower and upper bound on precision and recall. Furthermore, the manually investigated sample may not generalise to the entire studied dataset. We mitigate this threat by applying established filtering criteria when constructing the dataset, choosing a statistically significant sample size, and sampling from a deduplicated pool to avoid studying duplicates.

A threat to *construct* validity stems from the reliance of the Software Composition Analysis on instances of dependency management implementation patterns as a proxy for dependencies. The SCA and Joern, the framework on which it is built, may suffer from technical bugs and limitations (cf. Section 6.4.4) which may hamper the correct identification of dependencies. We mitigated this threat by evaluating the SCA's precision and recall in RQ_3, which showed it is accurate. Moreover, we omitted the *ansible.builtin* collection to avoid studying Ansible's own dependencies.

As a threat to *external validity*, our study focuses only on Ansible. Nonetheless, we expect that similar observations can be made in other IaC tools. For instance, Chef and Puppet, which are configuration management languages similar to Ansible, may exhibit similar properties in the dependency networks of their plugins. PL-IaC frameworks such as Pulumi provide APIs to be used in general-purpose programming languages and may therefore reuse the language's existing dependency ecosystem. Moreover, because Pulumi focuses on provisioning of cloud infrastructures from pre-built templates, it is heavily dependent on their remote APIs, but may be less dependent on OS packages. Containerisation tools such as Docker have naturally extensive supply chains, comprising many OS packages and development libraries [148], and other Docker images through inheritance [88]. Orchestration frameworks like Kubernetes depend on those Docker images, and tools to manage Kubernetes manifests, like Helm, may depend on other content in their own

ecosystem [152]. A major difference is that OS packages and development libraries in Docker images are primarily run-time dependencies [148], whereas for Ansible plugins, the dependencies are required at deployment time.

6.5 Related Work

In this section, we briefly introduce previous work related to our empirical study. First, we focus on prior studies on software dependencies, both for run-time and development dependencies (Section 6.5.1) and deployment dependencies (Section 6.5.2). Afterwards, we review existing approaches to identify dependencies from source code (Section 6.5.3).

6.5.1 Run-time and Development Dependencies

Many studies have investigated dependencies that are necessary at run time or during development. These studies typically focus on the ecosystems formed around dependency management tools. For instance, the NPM ecosystem for JavaScript has been widely studied, often in a security context [2, 20, 29, 72, 114, 139, 146]. Other work has focused on ecosystems like Java's Maven [48, 53, 99], Python's PyPI [42, 137, 144], C/C++ [142], OS packages [6], or comparing common properties across multiple ecosystems [1, 28, 63, 149].

Other researchers have instead studied dependencies in specific software domains. Huang et al. [52] study dependency-related bugs in Deep Learning stacks, whose software and hardware dependencies are diverse. Fang et al. [37] identify dependency antipatterns in distributed microservice architectures from dependency information originating from different sources, employing build dependencies extracted from dependency manifests and run-time dependencies from dynamic call traces. Our work differs from these existing studies as we investigate deployment dependencies in IaC, for which no structured manifests exist.

6.5.2 Deployment Dependencies

In contrast to the work on run-time and development dependencies outlined above, recent work has investigated deployment dependencies, which typically manifest themselves in Infrastructure-as-Code specifications.

For instance, the ecosystem of Docker images has been the subject of several empirical studies. Zhao et al. [155] investigate Docker Hub's image characteristics, such as layer sizes. Zerouali et al. investigate the use of run-time dependencies from ecosystems like NPM, PyPI, and Debian packages in Docker images [147, 148, 150, 151], focusing on security vulnerabilities and outdatedness. Lin et al. [70] study the evolution of Docker files and their corresponding images on Docker Hub. Ibrahim et al. [56] compare characteristics such as image size between official and community images on Docker Hub. We have previously investigated deployment dependencies in Docker images that result from inheritance between images [88]. Finally, Zerouali et al. [152] investigate the

ecosystem forming around Kubernetes manifests, specifically Helm charts on Artifact Hub, focusing on outdatedness and security vulnerabilities, and find that they often depend on multiple Docker images.

6.5.3 Dependency Identification

Many of the aforementioned studies rely on dependency manager manifest files to identify third-party software dependencies. However, some languages, notably C/C++, have no widely-used dependency management tools, posing a challenge to identify dependencies. Moreover, dependency manifests may not be available for software distributed in binary formats, e.g., mobile applications. Therefore, several earlier studies have investigated how dependencies can be identified for systems written in such languages.

Many of these prior approaches rely on code clone detection, either in source code [59, 73, 141, 143] or in compiled binaries [33, 124, 133, 145]. Tang et al. [134] combine such code clone detection techniques with information extracted from numerous package managers and build systems for C and C++. Our Software Composition Analysis differs from these existing dependency identification approaches as we do not rely on code clone detection and instead match dependency management patterns in the source code.

6.6 Conclusion

Understanding the software supply chains that support deployment code is vital to secure digital infrastructures. However, such deployment software supply chains have not been studied before. Therefore, in this chapter, we have presented a qualitative and quantitative empirical study into the dependencies of Ansible plugins.

We manually studied a statistically significant sample of 266 documented requirements and their implementations in Ansible plugins. We constructed a taxonomy of 7 types of third-party software that Ansible plugins may depend upon, finding that Python libraries and OS packages are the most common Ansible plugin dependencies. Moreover, we found that most studied plugins check that their dependencies are satisfied, and we have identified 5 common dependency management patterns in their implementation. Nonetheless, plugins shift the responsibility of installing these dependencies to the end users of the plugins.

Based on the dependency management patterns, we designed and implemented a Software Composition Analysis that extracts dependencies from Ansible plugin implementations. We show that our approach is accurate, achieving 61%–77% recall and 74%–95% precision on a ground truth of 221 plugin dependencies. By applying our extractor in a large-scale quantitative experiment involving 187 Ansible collections, we found that 51% of collections and 38% of plugins have dependencies. Most Ansible plugins depend only on a single type of dependency, yet end users of Ansible collections may need to

manage extensive and complex deployment software supply chains to ensure reproducibility of their deployment code. These supply chains may commonly consist of Ansible artefacts, Python packages, and OS libraries.

Our results suggest the need for improved dependency management practices for Ansible. Otherwise, deployment automation that creates reproducible environments may itself not be reproducible.

Chapter 7

Conclusion

Infrastructure as Code (IaC) enables practitioners to automate their software deployments through executable code. Although IaC can increase reliability and decrease human error, IaC artefacts are not safe from coding errors and security flaws. This thesis envisioned **sophisticated code quality assurance approaches for Ansible**, one of the most popular and versatile IaC tools, to aid practitioners in identifying and rectifying such issues, akin to those for application code.

We first reviewed the academic literature on quality assurance approaches for IaC. We identified three quality aspects and six analysis techniques that existing work has considered. From this review, we identified two knowledge gaps. The first is a lack of infrastructure code scanning approaches that utilise data-flow analyses, a lightweight yet accurate form of static analysis. The second revolves around software supply chains of deployment code, which, although vital to understand, have not been studied before.

To address the first knowledge gap, we proposed **two new static code representations for Ansible**. The first is a **syntactic representation of Ansible YAML code** which we called the *structural model*. It improves upon existing syntactic representations by integrating domain-specific knowledge into the parsing of YAML files, enabling it to better represent Ansible concepts. The second representation is the **Program Dependence Graph (PDG) for Ansible**. Its nodes represent program elements, such as actions, expressions, and data values, while its edges connect the elements through control-flow and data-flow relationships. The PDG constructor derives this information through a data-flow analysis, and thereby **enables sophisticated Ansible quality assurance approaches that employ data-flow information**.

We presented a first application of the PDG representation in the form of a **code smell detector that identifies error-prone coding patterns related to Ansible variables and expressions**. We introduced a catalogue of six such smells, inspired by real-world problems and intricate Ansible semantics. Our approach detects these smells by traversing the PDGs in search of these coding patterns. Using a prototypical implementation, we manually reviewed a random sample of 120 smells and showed that our approach is accurate, achieving a precision of 92%. We further used the prototype in a large-scale empirical study involving over 20 000 Ansible projects. This showed that the

smells are pervasive throughout the entire lifetime of projects. Moreover, we found that the smells may be indicative of defects, yet take a long time to be fixed.

As a second application of the PDGs, we proposed a **security smell detector for Ansible that improves upon state-of-the-art detectors by integrating control-flow and data-flow information**. The detector uses Cypher graph queries to identify 7 types of security flaws in the PDGs. Through an evaluation on a manually-constructed oracle of 243 real-world security flaws, we showed that our approach achieves high precision and recall and outperforms state-of-the-art detectors. We applied the detector in another large-scale empirical study, in which we found nearly 8 000 security smells across nearly 500 Ansible projects. Our findings showed that more than half of the smells need control-flow and data-flow information to be detected accurately. This further motivates the need to integrate control-flow and data-flow information into quality assurance tooling for IaC.

Finally, we addressed the second knowledge gap by conducting a **qualitative empirical study of the deployment software supply chain of Ansible collections**. By studying 266 requirements, we constructed a taxonomy of 7 types of software upon which plugins may depend, such as Python packages and operating system libraries. We then proposed a **Software Composition Analysis that extracts an Ansible plugin's dependencies** by matching dependency management patterns in the plugin's implementation. We showed that this approach is accurate using an oracle of 221 dependencies. Through a large-scale quantitative experiment, we observed that half of the collections and 38% of the plugins depend on other software. This indicated that the deployment software supply chain of Ansible automation code, which typically uses numerous collections and plugins, may be extensive.

7.1 Contributions, Revisited

This dissertation has made the following main contributions:

Program Dependence Graph for Ansible In Section 3.3, we proposed the Program Dependence Graph (PDG) representation for Ansible. The PDG is a novel static representation for Infrastructure as Code which succinctly models Ansible code's control-flow and data-flow information. Its nodes represent code elements such as Ansible actions, expressions, and values. They are interconnected through control-flow order and data-flow definition and use edges. We also described a static data-flow analysis to construct this PDG representation, highlighting some of Ansible's semantic peculiarities and how the data-flow analysis accounts for them. This representation forms the basis for sophisticated development tooling for Ansible.

Sophisticated smell detection approaches for Ansible We proposed, implemented, and evaluated two smell detection approaches based on this

PDG representation. First, in Sections 4.2 and 4.3 we described a catalogue of 6 variable-related code smells that are indicative of potential defects in Ansible infrastructure code, and a detector that traverses the PDG representation in search of these bad practices. Our evaluation on 120 randomly-sampled reports showed that this detector achieves high precision. Second, in Section 5.3, we described a smell detector for 7 security smells that are indicative of possible security weaknesses in Ansible code. This approach outperforms two state-of-the-art detectors in an evaluation on an oracle of 243 manually-reviewed weaknesses because of its use of data-flow information and its ability to represent jumps in control flow.

Combined, these smell detection approaches contribute to bridging the first knowledge gap we identified, namely the lack of lightweight yet accurate code scanning approaches for Infrastructure as Code. They may aid Ansible practitioners to enhance the security of the infrastructure scripts, or to identify potential defects early. Moreover, they show the importance of including semantic information in code scanning, and indicate that such inclusion is feasible in practical tooling.

Uncovering software supply chains of Ansible deployments We manually reviewed the dependencies of Ansible plugins to gain a better understanding of deployment software supply chains, a previously-unstudied topic (Section 6.2). From this, we have derived a taxonomy of the types of software that can occur in Ansible Infrastructure-as-Code deployment software supply chains. We found that these mainly consist of Python packages and OS binaries, yet may also include various other types of software. We also described a catalogue of 5 dependency management practices implemented by Ansible plugins. This taxonomy and catalogue may be useful to practitioners to implement dependency management in their own projects, for which Ansible does not provide utilities. Moreover, the study sheds light on an increasingly critical yet understudied topic in software security.

Software Composition Analysis for Ansible In Section 6.3, we proposed, implemented, and evaluated a Software Composition Analysis for Ansible that identifies the dependency management practices described in the previous contribution, through which it automatically extracts the dependencies of a plugin. We evaluated it on an oracle of 221 dependencies, achieving 61%–77% recall and 74%–95% precision. This analysis forms an important first step towards the automated generation of Software Bills of Materials for Ansible Infrastructure as Code. The automated extraction of dependencies could also be used in other contexts, such as the validation of documented requirements, or automated installation of dependencies in client projects.

Large-scale empirical studies into Ansible code quality Finally, we carried out three empirical studies in which we apply our approaches on a large scale to quantitatively investigate Ansible code quality in practice. In the

first (cf. Section 4.4), we studied the prevalence and lifetime of variable-related code smells in 21 931 open-source Ansible roles (Section 4.4). We found that these smells are increasingly pervasive and may remain in codebases for a long time, even though they may be indicative of defects. In the second empirical study (cf. Section 5.4), we investigated the prevalence of data-flow and control-flow indirections in security smells found across 15 000 Ansible scripts. We found that half of all detected smells involve some form of data flow, while a third involves control-flow jumps. In the final study (cf. Section 6.3), we automatically extracted the dependencies of 187 Ansible collections and their plugins to study deployment software supply chains in practice. We observed that substantial proportions of collections and plugins require other types of software, leading to extensive deployment supply chains. Combined, the results of these empirical studies provide empirically-substantiated motivations for future work to build upon. Moreover, the datasets curated in these empirical studies have been made available in their respective replication packages, enabling future researchers to reuse them.

7.2 Limitations of the Approaches

In this section, we summarise the main limitations exhibited by our approaches.

Technical Limitations Our prototypical implementations may suffer from technical limitations. For instance, the Program Dependence Graph builder cannot reliably handle dynamic inclusions of files whose name is specified as an expression, and instead ignores such file inclusions. Moreover, it disregards conditional variable definitions, which may lead to incorrect approximations in certain cases. Such limitations also impact the smell detection approaches that build upon the PDGs. Nonetheless, we did not observe a substantial negative impact of these limitations in our evaluations. However, for practical implementations, these limitations could be fixed by using a more advanced data-flow analysis.

Similarly, the smell detection approaches may suffer from their own technical limitations. Our evaluation showed that the security smell detector achieves relatively low precision and recall for certain types of security smells due to flaws in its string patterns. In particular, the string patterns used to identify secrets, such as passwords, are ineffective and often produced false positives. Future work could therefore investigate means through which these false positives can be eliminated.

Finally, the Software Composition Analysis relies on implementation patterns to identify dependencies. However, such patterns are not always present, causing the SCA to miss certain dependencies. Moreover, several dependency types are checked ad hoc, or not checked at all, which also cannot be found by the SCA. Therefore, future work should investigate whether such dependencies can be identified through other mechanisms.

Lack of Practitioner Feedback We did not consult with Ansible practitioners when designing and implementing the approaches presented in this dissertation. Nonetheless, throughout our empirical studies, we found anecdotal evidence in the form of bug-fixing commits that indicate that our smell catalogues and detection approaches may be useful for practitioners. However, existing work has shown that practitioner feedback can prove vital in improving smell detection approaches [113]. Therefore, future work should consult with practitioners to integrate their feedback into practical tooling based on our approaches. Moreover, practitioners' feedback may provide inspiration to extend our smell catalogues, which are not exhaustive.

Generalisability to Other IaC Languages Our approaches have been instantiated for Ansible alone, and we did not investigate whether they can be applied in other IaC languages. We discuss the generalisability of our approaches in more detail in the next section. However, Ansible is one of the most widely-used IaC languages [40, 130]. Our approaches may form the basis for practical tooling for its practitioners, and our findings could guide improvements to the Ansible language.

7.3 Generalisability of the Approaches

The approaches presented in this dissertation are designed and implemented for Ansible. As such, although our approaches are tailored specifically to Ansible, certain parts may generalise to the wider Infrastructure-as-Code domain. In this section, we discuss the generalisability of our work from three perspectives, namely the *concepts and ideas* presented in this dissertation, the *approaches and algorithms*, and the concrete *empirical study results*.

Concepts and Ideas We argue that the majority of the concepts and ideas presented in this dissertation can be applied in the broader IaC domain. Static data-flow analysis, as described in Chapter 3, could be applied to other configuration management IaC languages, such as Puppet and Chef, to obtain similar approaches to those presented in this dissertation. Moreover, it could be applied to provisioning IaC languages, e.g., Terraform, to collect information on how data is defined and used in provisioning specifications. Such information could later be used to check whether cloud resources are configured with defect-prone or insecure values. Similarly, representations akin to our Program Dependence Graphs could be generated for other IaC languages to enable more sophisticated software quality assurance approaches.

Furthermore, behavioural code smell detection, described in Chapter 4, and behavioural security smell detection, described in Chapter 5, could also be applied to other IaC languages. Prior work has already shown the feasibility of detecting code smells [117, 120] and security smells [106, 108, 109] in other IaC languages. Moreover, smells can be detected in a polyglot setting, in which a single detector can scan IaC specifications written in different languages [115, 116]. This provides confidence that integrating behavioural information into code smell and security smell detection is feasible for the broader IaC domain.

Finally, while our study of deployment software supply chains focused solely on Ansible (cf. Chapter 6), this study could be replicated to other IaC languages to understand how their artefacts use third-party software. We believe similar taxonomies of dependencies could be derived, and that using implementation patterns to uncover these dependencies could serve as a viable approach to study software supply chains for other IaC languages. Furthermore, the idea of matching implementation patterns to identify dependencies may be applicable outside the IaC domain as well, in cases where structured manifests are not available.

Approaches and Algorithms While the approaches and algorithms described in this dissertation are tailored to Ansible, we believe that several could be generalised to other IaC languages with minor adjustments.

The structure of the PDG representation may require minor changes to support other configuration management languages. Notably, to represent Puppet code, which uses a non-deterministic execution order, the control-flow order edges represented in the PDG would have to be adapted to form a partial order. Nonetheless, we believe the overall structure of the PDGs would lend themselves well to represent other IaC languages. Naturally, the procedure to build such PDGs would need to be adjusted to support a new language, which may also require accounting for the language's semantics.

Once the PDGs are obtained, our smell detection approaches could readily be applied, yet minor changes may be necessary (e.g., to incorporate language-specific information in the string patterns used by the security smell queries). Note, however, that the variable-related code smells, which are inspired specifically by Ansible's semantics, may not be applicable to all IaC languages.

Empirical Study Results As our empirical studies considered only Ansible code, we cannot claim that they generalise to the entire IaC domain. Nonetheless, if one implements the necessary changes to the approaches as described above, our studies could be replicated to other IaC languages and their results could be compared. Prior work has already compared the prevalence of security smells between different configuration management languages using syntactic analyses [108, 115], yet as we showed in Section 5.4, we observe differences in the prevalences when applying data-flow analysis, motivating further comparative studies.

Finally, we hypothesise that the taxonomy of dependencies on third-party software presented in Section 6.2 may largely generalise to the broader IaC domain. Specifically, we believe that similar dependencies may be found in other configuration management IaC tools, such as Chef and Puppet. Moreover, provisioning tools (e.g., Terraform, Pulumi) likely make extensive use of remote APIs offered by cloud providers and programming libraries that offer wrappers around those APIs, both of which are types of software contained in our taxonomy.

7.4 Future Work

We discuss some potential avenues of future work.

Deployment Software Supply Chains The work presented in Chapter 6 provides numerous opportunities for future work. One direction would be to use the Software Composition Analysis to generate a Software Bill of Materials (SBOM) for the infrastructure code. This could then be used to further secure the deployed application by augmenting its own SBOM. Another direction would be to use our empirical findings to guide improvements to dependency management practices in IaC. For instance, we envision novel dependency management tools that are tailored to the unique properties exhibited by IaC dependencies, or improved language integration. Finally, our work only considered a single aspect of deployment software supply chains, namely the dependencies of Ansible plugins. However, the deployment software supply chain extends far beyond those, e.g., the transitive dependencies of those covered in Chapter 6, dependencies on Ansible roles [94], etc. Therefore, an avenue of future work could be to extend our empirical investigation to gain a more holistic understanding of deployment software supply chains.

Integration With Data-Driven Approaches We envision two opportunities to integrate data-driven approaches (e.g., machine learning) into our work. First, one could integrate data-driven approaches into our smell detectors. For instance, a machine learning model could be used to predict when an Ansible action argument is security-sensitive to improve the performance of the *hardcoded secret* smells detection in Section 5.3. Second, our PDG representation could be integrated into data-driven defect prediction approaches (cf. Section 2.3.2). We posit that using the PDGs and the semantic information they contain may improve the performance of the predictive models. Alternatively, the PDGs could be mined using graph pattern mining algorithms. The obtained patterns could be used to generate examples, identify possible defects through anomaly detection [3], or for program comprehension purposes [87].

Cross-layer Analysis Infrastructure-as-Code programs can be divided into an application layer and a runtime layer, the former containing the infrastructure specifications and the latter implementing the interactions with the targeted platforms [32]. Our approaches target both layers individually, with the PDG representation (Chapter 3) and smell detectors (Chapters 4 and 5) targeting the application layer, and the SCA (Chapter 6) targeting the runtime layer. Future work could investigate cross-layer analyses that consider the interactions between the application-layer specifications and the runtime-layer implementations. For instance, our security smell detection approach could be extended to analyse Python plugin implementations to identify security-sensitive parameters, which it can then use to improve the search for hardcoded secrets. Moreover, one could extend our Software Composition Analysis to also consider application-layer dependencies (e.g., on Ansible roles) to obtain a holistic overview of an Ansible project's deployment software supply chain.

Refactoring and Repair In Section 4.4, we found that code smells are introduced more frequently than they are fixed. This calls for automated means to repair and remediate these smells. We believe our representations could be used to this end. For instance, the structural model could be used to specify and execute code transformations and rewrite the original source code, while the PDG representation could be used to guide the search process for automated repairs. Moreover, we envision automated refactoring tools using the PDG representation to verify that a transformation is behaviour-preserving by comparing PDGs.

Other IaC Languages The approaches presented in this dissertation have all been instantiated for Ansible. We believe their core concepts could be replicated to other IaC languages (cf. Section 7.3). For instance, Chef, Puppet, and Salt, all configuration management languages closely related to Ansible, can likely benefit from similar approaches. Future work could extend our work to polyglot tooling, e.g., by adapting the PDG representation to suit other languages [115]. Several concepts are likely also applicable to other IaC languages, such as provisioning tools (e.g., Terraform), orchestrators (e.g., Kubernetes), and server templating tools (e.g., Packer). Future work may investigate the extent to which our approaches can be applied in other types of IaC.

7.5 Closing Remarks

We commenced this dissertation by positing that although Infrastructure as Code offers numerous benefits, the code implementing the automation can be plagued by defects and security weaknesses. We stated that there is a need for sophisticated tooling to aid practitioners in performing quality assurance of their infrastructure code. This dissertation has presented multiple novel approaches to fulfil that need for Ansible, one of the most popular and versatile IaC tools. The Program Dependence Graph representation for Ansible bridges the gap between lightweight yet inaccurate syntactic approaches and expensive and impractical formal verification by providing a lightweight representation that encodes an Ansible script's behaviour. Two smell detectors employing this representation, one for code smells indicating defects and one for security smells indicating weaknesses, enable sophisticated code scanners for Ansible. A Software Composition Analysis enables automated extraction and inspection of Ansible code's dependencies, shedding light on an underexplored concept: deployment software supply chains. Finally, numerous large-scale empirical studies conducted with prototypical implementations of these approaches not only showed their practical applicability, but also motivate widespread practical adoption of quality assurance tooling.

Bibliography

[1] Rabe Abdalkareem, Vinicius Oda, Suhaib Mujahid, and Emad Shihab. "On the impact of using trivial packages: An empirical case study on npm and PyPI". In: *Empirical Software Engineering* 25.2 (Sept. 2020). Ed. by Arie van Deursen, pp. 1168–1204. ISSN: 1573-7616. DOI: 10.1007/S10664-019-09792-9.

[2] Mahmoud Alfadel, Diego Elias Costa, Emad Shihab, and Bram Adams. "On the Discoverability of npm Vulnerabilities in Node.js Projects". In: *ACM Transactions on Software Engineering and Methodology* 32.4, 91 (July 2023). Ed. by Mauro Pezzè, 27 pp. ISSN: 1049-331X. DOI: 10.1145/3571848.

[3] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. "Investigating next Steps in Static API-Misuse Detection". In: *Proceedings of the 16th IEEE/ACM International Conference on Mining Software Repositories*. MSR 2019 (Montreal, Canada, May 26–27, 2019). Ed. by Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc. Los Alamitos, CA, USA: IEEE, 2019, pp. 265–275. ISBN: 978-1-7281-3412-3. DOI: 10.1109/MSR.2019.00053.

[4] Iosif Andrei-Cristian, Tiago Espinha Gasiba, Tiange Zhao, Ulrike Lechner, and Maria Pinto-Albuquerque. "A Large-Scale Study on the Security Vulnerabilities of Cloud Deployments". In: *Ubiquitous Security – First International Conference, Revised Selected Papers*. UbiSec 2021 (Guangzhou, China, Dec. 28–31, 2021). Ed. by Guojun Wang, Kim-Kwang Raymond Choo, Ryan K. L. Ko, Yang Xu, and Bruno Crispo. Communications in Computer and Information Science 1557. Berlin, Germany: Springer, 2021, pp. 171–188. ISBN: 978-981-19-0467-7. DOI: 10.1007/978-981-19-0468-4_13.

[5] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William W. Pugh. "Using Static Analysis to Find Bugs". In: *IEEE Software* 25.5 (Sept. 2008), pp. 22–29. ISSN: 0740-7459. DOI: 10.1109/MS.2008.130.

[6] Rahul Bajaj, Eduardo Fernandes, Bram Adams, and Ahmed E. Hassan. "Unreproducible builds: time to fix, causes, and correlation with external ecosystem factors". In: *Empirical Software Engineering* 29.1, 11 (Jan. 2024). Ed. by Philipp Leitner, 48 pp. ISSN: 1573-7616. DOI: 10.1007/S10664-023-10399-4.

[7] Mahi Begoug, Narjes Bessghaier, Ali Ouni, Eman Abdullah AlOmar, and Mohamed Wiem Mkaouer. "What Do Infrastructure-as-Code Practitioners Discuss: An Empirical Study on Stack Overflow". In: *Proceedings of the 2023 ACM/IEEE International Symposium on Empirical*

Software Engineering and Measurement. ESEM 2023 (New Orleans, LA, USA, Oct. 26–27, 2023). Ed. by Jeffrey Carver, Clemente Izurieta, Per Runeson, and Maleknaz Nayebi. Los Alamitos, CA, USA: IEEE, 2023, pp. 1–12. ISBN: 978-1-6654-5223-6. DOI: 10.1109/ESEM56168.2023.10304847.

[8] Mahi Begoug, Moataz Chouchen, Ali Ouni, Eman Abdullah Alomar, and Mohamed Wiem Mkaouer. "Fine-Grained Just-In-Time Defect Prediction at the Block Level in Infrastructure-as-Code (IaC)". In: *Proceedings of the 21st International Conference on Mining Software Repositories*. MSR 2024 (Lisbon, Portugal, Apr. 15–May 16, 2024). Ed. by Diomidis Spinellis, Alberto Bacchelli, and Eleni Constantinou. New York, NY, USA: ACM, 2024, pp. 100–112. ISBN: 979-8-4007-0587-8. DOI: 10.1145/3643991.3644934.

[9] Julian Bellendorf and Zoltán Ádám Mann. "Specification of cloud topologies and orchestration using TOSCA: a survey". In: *Computing* 102.8 (Aug. 2020). Ed. by Schahram Dustdar, pp. 1793–1815. DOI: 10.1007/S00607-019-00750-3.

[10] Narjes Bessghaier, Mohammed Sayagh, Ali Ouni, and Mohamed Wiem Mkaouer. "What Constitutes the Deployment and Runtime Configuration System? An Empirical Study on OpenStack Projects". In: *ACM Transactions on Software Engineering and Methodology* 33.1, 5 (Jan. 2024). Ed. by Mauro Pezzè, 37 pp. ISSN: 1049-331X. DOI: 10.1145/3607186.

[11] Farzana Ahamed Bhuiyan and Akond Rahman. "Characterizing co-located insecure coding patterns in infrastructure as code scripts". In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2020 (Virtual Event, Melbourne, Australia, Sept. 22–25, 2020). Ed. by John Grundy, Claire Le Goues, and David Lo. New York, NY, USA: ACM, 2020, pp. 27–32. ISBN: 978-1-4503-6768-4. DOI: 10.1145/3417113.3422154.

[12] Nemania Borovits, Indika Kumara, Dario Di Nucci, Parvathy Krishnan, Stefano Dalla Palma, Fabio Palomba, Damian Andrew Tamburri, and Willem-Jan van den Heuvel. "FindICI: Using machine learning to detect linguistic inconsistencies between code and natural language descriptions in infrastructure-as-code". In: *Empirical Software Engineering* 27.7, 178 (Dec. 2022): *Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. Ed. by Foutse Khomh, Gemma Catolino, and Pasquale Salza, 41 pp. ISSN: 1573-7616. DOI: 10.1007/s10664-022-10131-8.

[13] Nemania Borovits, Indika Kumara, Parvathy Krishnan, Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, Damian Andrew Tamburri, and Willem-Jan 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*. MaLTeSQuE 2020 (Virtual Event, USA, Nov. 13, 2020). Ed. by Foutse Khomh, Pasquale Salza, and Gemma Catolino. New York, NY, USA: ACM, 2020, pp. 7–12. ISBN: 978-1-4503-8124-6. DOI: 10.1145/3416505.3423564.

[14] Yevgeniy Brikman. *Terraform: Up and Running*. 3rd ed. Sebastopol, CA, USA: O'Reilly, 2022. ISBN: 978-1-098-11674-3.

[15] Antonio Brogi, Antonio Di Tommaso, and Jacopo Soldani. "Sommelier: A Tool for Validating TOSCA Application Topologies". In: *Model-Driven Engineering and Software Development – 6th International Conference, Revised Selected Papers*. MODELSWARD 2018 (Funchal, Madeira, Portugal, Jan. 22–24, 2018). Ed. by Luís Ferreira Pires, Slimane Hammoudi, and Bran Selic. Berlin, Germany: Springer, 2018, pp. 1–22. ISBN: 978-3-319-94764-8. DOI: 10.1007/978-3-319-94764-8_1.

[16] Claudia Cauli, Meng Li, Nir Piterman, and Oksana Tkachuk. "Predeployment Security Assessment for Cloud Services Through Semantic Reasoning". In: *Computer Aided Verification – 33rd International Conference, Proceedings, Part I*. CAV 2021 (Virtual Event, Los Angeles, CA, USA, July 20–23, 2021). Ed. by Alexandra Silva and K. Rustan M. Leino. Lecture Notes in Computer Science 12759. Berlin, Germany: Springer, 2021, pp. 767–780. ISBN: 978-3-030-81684-1. DOI: 10.1007/978-3-030-81685-8_36.

[17] Wei Chen, Guoquan Wu, and Jun Wei. "An Approach to Identifying Error Patterns for Infrastructure as Code". In: *Proceedings of the 29th IEEE International Symposium on Software Reliability Engineering Workshops*. Industry Track. ISSREW 2018 (Memphis, TN, USA, Oct. 15–18, 2018). Ed. by Bojan Cukic, Robin S. Poston, Catello Di Martino, Harigovind Ramasamy, and Mike Sayres. Los Alamitos, CA, USA: IEEE, 2018, pp. 124–129. ISBN: 978-1-5386-9443-5. DOI: 10.1109/ISSREW.2018.00-19.

[18] Michele Chiari, Michele De Pascalis, and Matteo Pradella. "Static Analysis of Infrastructure as Code: a Survey". In: *Companion Proceedings of the 19th IEEE International Conference on Software Architecture*. 1st International Workshop on the Foundations of Infrastructure Specification and Testing (FIST 2022) (Honolulu, HI, USA, Mar. 12, 2022). Ed. by Luciano Baresi, Giovanni Quattrocchi, and Damian Andrew Tamburri. Los Alamitos, CA, USA: IEEE, 2022, pp. 218–225. ISBN: 978-1-6654-1728-0. DOI: 10.1109/ICSA-C54293.2022.00049.

[19] Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong. "Orthogonal Defect Classification — A Concept for In-Process Measurements". In: *IEEE Transactions on Software Engineering* 18.11 (Nov. 1992). Ed. by Richard Selby and Koji Torii, pp. 943–956. ISSN: 0098-5589. DOI: 10.1109/32.177364.

[20] Md Atique Reza Chowdhury, Rabe Abdalkareem, Emad Shihab, and Bram Adams. "On the Untriviality of Trivial Packages: An Empirical Study of npm JavaScript Packages". In: *IEEE Transactions on Software Engineering* 48.8 (Aug. 2022). Ed. by Patrick Eugster, pp. 2695–2708. ISSN: 0098-5589. DOI: 10.1109/TSE.2021.3068901.

[21] William Gemmell Cochran. *Sampling Techniques*. 3rd ed. New York, NY, USA: John Wiley & Sons, 1977. ISBN: 0-471-02939-4.

[22] Ting Dai, Alexei Karve, Grzegorz Koper, and Sai Zeng. "Automatically Detecting Risky Scripts in Infrastructure Code". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. SoCC 2020 (Virtual Event, USA, Oct. 19–21, 2020). Ed. by Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi. New York, NY, USA: ACM, 2020, pp. 358–371. ISBN: 978-1-4503-8137-6. DOI: 10.1145/3419111.3421303.

[23] Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, and Damian Andrew Tamburri. "Toward a catalog of software quality metrics for infrastructure code". In: *Journal of Systems and Software* 170, 110726 (Dec. 2020): *New Trends and Ideas*. Ed. by Paris Avgeriou and David Shepherd, 8 pp. ISSN: 0164-1212. DOI: 10.1016/j.jss.2020.110726.

[24] Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, and Damien Andrew Tamburri. "Within-Project Defect Prediction of Infrastructure-as-Code Using Product and Process Metrics". In: *IEEE Transactions on Software Engineering* 48.6 (June 2022). Ed. by Leonardo Mariani, pp. 2086–2104. ISSN: 0098-5589. DOI: 10.1109/TSE.2021.3051492.

[25] Stefano Dalla Palma, Dario Di Nucci, and Damian Andrew Tamburri. "AnsibleMetrics: A Python library for measuring Infrastructure-as-Code blueprints in Ansible". In: *SoftwareX* 12, 100633 (July 2020). Ed. by Kate Keahey, Frank Seinstra, and David Wallom, 6 pp. ISSN: 2352-7110. DOI: 10.1016/j.softx.2020.100633.

[26] Stefano Dalla Palma, Majid Mohammadi, Dario Di Nucci, and Damian Andrew 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*. MaLTeSQuE 2020 (Virtual Event, USA, Nov. 13, 2020). Ed. by Foutse Khomh, Pasquale Salza, and Gemma Catolino. New York, NY, USA: ACM, 2020, pp. 31–36. ISBN: 978-1-4503-8124-6. DOI: 10.1145/3416505.3423563.

[27] Stefano Dalla Palma, Chiel van Asseldonk, Gemma Catolino, Dario Di Nucci, Fabio Palomba, and Damian Andrew Tamburri. "'"Through the looking-glass ..." An empirical study on blob infrastructure blueprints in the Topology and Orchestration Specification for Cloud Applications". In: *Journal of Software: Evolution and Process* 36.4, e2533 (Apr. 2024). Ed. by Darren Dalcher, David Raffo, Massimiliano Di Penta, and Xin Peng, 19 pp. ISSN: 2047-7473. DOI: 10.1002/SMR.2533.

[28] Alexandre Decan, Tom Mens, and Maëlick Claes. "An empirical comparison of dependency issues in OSS packaging ecosystems". In: *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering*. SANER 2017 (Klagenfurt, Austria, Feb. 20–24, 2017). Ed. by Martin Pinzger, Gabriele Bavota, and Andrian Marcus. Los Alamitos, CA, USA: IEEE, 2017, pp. 2–12. ISBN: 978-1-5090-5501-2. DOI: 10.1109/SANER.2017.7884604.

[29] Alexandre Decan, Tom Mens, and Eleni Constantinou. "On the impact of security vulnerabilities in the npm package dependency network". In: *Proceedings of the 15th ACM/IEEE International Conference on Mining Software Repositories*. MSR 2018 (Gothenburg, Sweden, May 28–29,

2018). Ed. by Andy Zaidman, Yasutaka Kamei, and Emily Hill. New York, NY, USA: ACM, 2018, pp. 181–191. ISBN: 978-1-4503-5716-6. DOI: 10.1145/3196398.3196401.

[30] Alexandre Decan, Tom Mens, and Hassan Onsori Delicheh. "On the outdatedness of workflows in the GitHub Actions ecosystem". In: *Journal of Systems and Software* 206, 111827 (Dec. 2023). Ed. by Shane McIntosh, 20 pp. ISSN: 0164-1212. DOI: 10.1016/j.jss.2023.111827.

[31] Giorgio Dell'Immagine, Jacopo Soldani, and Antonio Brogi. "KubeHound: Detecting Microservices' Security Smells in Kubernetes Deployments". In: *Future Internet* 15.7, 228 (2023): *Information and Future Internet Security, Trust and Privacy II*. Ed. by Weizhi Meng and Christian D. Jensen, 26 pp. ISSN: 1999-5903. DOI: 10.3390/fi15070228.

[32] Georgios-Petros Drosos, Thodoris Sotiropoulos, Georgios Alexopoulos, Dimitris Mitropoulos, and Zhendong Su. "When Your Infrastructure is a Buggy Program: Understanding Faults in Infrastructure as Code Ecosystems". In: *Proceedings of the ACM on Programming Languages* 8, 359 (OOPSLA2 Oct. 2024). Ed. by Michael Hicks, 31 pp. ISSN: 2475-1421. DOI: 10.1145/3689799.

[33] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. "Identifying Open-Source License Violation and 1-day Security Risk at Large Scale". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS 2017 (Dallas, TX, USA, Oct. 30–Nov. 3, 2017). Ed. by Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. New York, NY, USA: ACM, 2017, pp. 2169–2185. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134048.

[34] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. 1st ed. Boston, MA, USA: Addison-Wesley, 2007. ISBN: 978-0-321-33638-5.

[35] Tiago Espinha Gasiba, Andrei-Cristian Iosif, Ulrike Lechner, and Maria Pinto-Albuquerque. "Raising Security Awareness of Cloud Deployments using Infrastructure as Code through CyberSecurity Challenges". In: *Proceedings of the 16th International Conference on Availability, Reliability and Security*. ARES 2021 (Vienna, Austria, Aug. 17–20, 2021). Ed. by Delphine Reinhardt and Tilo Müller. New York, NY, USA: ACM, 2021, 63, 8 pp. ISBN: 978-1-4503-9051-4. DOI: 10.1145/3465481.3470030.

[36] Ghareeb Falazi, Lukas Harzenetter, Kálmán Képes, Frank Leymann, Uwe Breitenbücher, Evangelos Ntentos, Uwe Zdun, Martin Becker, and Elena Heldwein. "Compliance Management of IaC-Based Cloud Deployments During Runtime". In: *Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing*. UCC 2023 (Taormina (Messina), Italy, Dec. 4–7, 2023). Ed. by Massimo Villari, Omer Rana, Song Fu, and Lorenzo Carnevale. New York, NY, USA: ACM, 2023, 10, 11 pp. ISBN: 979-8-4007-0234-1. DOI: 10.1145/3603166.3632135.

[37] Hongzhou Fang, Yuanfang Cai, Rick Kazman, and Jason Lefever. "Identifying Anti-Patterns in Distributed Systems With Heterogeneous

Dependencies". In: *Companion Proceedings of the 20th International Conference on Software Architecture*. New and Emerging Ideas Track. ICSA-C 2023 (L'Aquila, Italy, Mar. 13–17, 2023). Ed. by Matthias Galster and J. Andrés Diaz-Pace. Los Alamitos, CA, USA: IEEE, 2023, pp. 116–120. ISBN: 978-1-6654-6459-8. DOI: 10.1109/ICSA-C57050.2023.00035.

[38] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The Program Dependence Graph and Its Use in Optimization". In: *ACM Transactions on Programming Languages and Systems* 9.3 (July 1987). Ed. by Susan L. Graham, pp. 319–349. ISSN: 0164-0925. DOI: 10.1145/24039.24041.

[39] Weili Fu, Roly Perera, Paul Anderson, and James Cheney. "μPuppet: A Declarative Subset of the Puppet Configuration Language". In: *Proceedings of the 31st European Conference on Object-Oriented Programming*. ECOOP 2017 (Barcelona, Spain, June 19–23, 2017). Ed. by Peter Müller. Leibniz International Proceedings in Informatics 74. Wadern, Germany: Dagstuhl Publishing, 2017, 12, 27 pp. ISBN: 978-3-95977-035-4. DOI: 10.4230/LIPICS.ECOOP.2017.12.

[40] Michele Guerriero, Martin Garriga, Damian Andrew Tamburri, and Fabio Palomba. "Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry". In: *Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution*. ICSME 2019 (Cleveland, OH, USA, Sept. 30–Oct. 4, 2019). Ed. by Jonathan I. Maletic, Brian Robinson, Miryung Kim, and Árpád Beszédes. Los Alamitos, CA, USA: IEEE, 2019, pp. 580–589. ISBN: 978-1-7281-3094-1. DOI: 10.1109/ICSME.2019.00092.

[41] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. "Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages". In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. SoCC 2016 (Santa Clara, CA, USA, Oct. 5–7, 2016). Ed. by Marcos K. Aguilera, Brian Cooper, and Yanlei Diao. New York, NY, USA: ACM, 2016, pp. 1–16. ISBN: 978-1-4503-4525-5. DOI: 10.1145/2987550.2987583.

[42] Wenbo Guo, Zhengzi Xu, Chengwei Liu, Cheng Huang, Yong Fang, and Yang Liu. "An Empirical Study of Malicious Code In PyPI Ecosystem". In: *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2023 (Echternach, Luxembourg, Sept. 11–15, 2023). Ed. by Tegawendé F. Bissyandé, Jacques Klein, Christian Bird, and Federica Sarro. Los Alamitos, CA, USA: IEEE, 2023, pp. 166–177. ISBN: 979-8-3503-2996-4. DOI: 10.1109/ASE56229.2023.00135.

[43] Christian Hammer and Gregor Snelting. "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs". In: *International Journal of Information Security* 8.6 (Dec. 2009), pp. 399–422. ISSN: 1615-5270. DOI: 10.1007/s10207-009-0086-1.

[44] Oliver Hanappi, Waldemar Hummer, and Schahram Dustdar. "Asserting reliable convergence for configuration management scripts". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on*

Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2016 (Amsterdam, Netherlands, Oct. 30–Nov. 4, 2016). Ed. by Eelco Visser and Yannis Smaragdakis. New York, NY, USA: ACM, 2016, pp. 328–343. ISBN: 978-1-4503-4444-9. DOI: 10.1145/2983990.2984000.

[45] Mohammed Mehedi Hasan, Farzana Ahamed Bhuiyan, and Akond Rahman. "Testing practices for infrastructure as code". In: *Proceedings of the 1st ACM SIGSOFT International Workshop on Languages and Tools for Next-Generation Testing*. LANGETI 2020 (Virtual Event, USA, Nov. 8–9, 2020). Ed. by Nicolás Cardozo, Ivana Dusparic, Mario Linares-Vásquez, Kevin Moran, and Camilo Escobar-Velásquez. New York, NY, USA: ACM, 2020, pp. 7–12. ISBN: 978-1-4503-8123-9. DOI: 10.1145/3416504.3424334.

[46] Md. Mahadi Hassan, John Salvador, Shubhra Kanti Karmaker Santu, and Akond Rahman. "State Reconciliation Defects in Infrastructure as Code". In: *Proceedings of the ACM on Software Engineering* 1, 83 (FSE July 2024). Ed. by Luciano Baresi, David Lo, and Lin Tan, pp. 1865–1888. ISSN: 2994-970X. DOI: 10.1145/3660790.

[47] Mohammad Mehedi Hassan and Akond Rahman. "As Code Testing: Characterizing Test Quality in Open Source Ansible Development". In: *Proceedings of the 15th IEEE International Conference on Software Testing, Verification and Validation*. ICST 2022 (Virtual Event, Valencia, Spain, Apr. 4–13, 2022). Ed. by Tanja E.J. Vos, Alessandra Gorla, Fitsum M. Kifetew, and Annibale Panichella. Los Alamitos, CA, USA: IEEE, 2022, pp. 208–219. ISBN: 978-1-6654-6679-0. DOI: 10.1109/ICST53961.2022.00031.

[48] Hao He, Runzhi He, Haiqiao Gu, and Minghui Zhou. "A large-scale empirical study on Java library migrations: prevalence, trends, and rationales". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021 (Athens, Greece, Aug. 23–28, 2021). Ed. by Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta. New York, NY, USA: ACM, 2021, pp. 478–490. ISBN: 978-1-4503-8562-6. DOI: 10.1145/3468264.3468571.

[49] Eric Horton and Chris Parnin. "Dozer: Migrating Shell Commands to Ansible Modules via Execution Profiling and Synthesis". In: *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP 2022 (Pittsburgh, PA, USA, May 22–27, 2022). Ed. by Matthew B. Dwyer, Mark Harman, and Heather Miller. Los Alamitos, CA, USA: IEEE, 2022, pp. 147–148. ISBN: 978-1-6654-9590-5. DOI: 10.1145/3510457.3513060.

[50] Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. "Identifying, Tailoring, and Suggesting Form Template Method Refactoring Opportunities with Program Dependence Graph". In: *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*. CSMR 2012 (Szeged, Hungary, Mar. 27–30, 2012). Ed. by Tom Mens, Anthony Cleve, and Rudolf Ferenc. Los Alamitos, CA, USA: IEEE, 2012, pp. 53–62. ISBN: 978-1-4673-0984-4. DOI: 10.1109/CSMR.2012.16.

[51] Hanyang Hu, Yani Bu, Kristen Wong, Gaurav Sood, Karen Smiley, and Akond Rahman. "Characterizing Static Analysis Alerts for Terraform Manifests: An Experience Report". In: *Proceedings of the 2023 IEEE Secure Development Conference*. SecDev 2023 (Atlanta, GA, USA, Oct. 8–10, 2023). Ed. by Tuba Yavuz, Eric Bodden, Leigh Metcalf, and Raghudeep Kannavara. Los Alamitos, CA, USA: IEEE, 2023, pp. 7–13. ISBN: 979-8-3503-3132-5. DOI: 10.1109/SecDev56634.2023.00014.

[52] Kaifeng Huang, Bihuan Chen, Susheng Wu, Junming Cao, Lei Ma, and Xin Peng. "Demystifying Dependency Bugs in Deep Learning Stack". In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2023 (San Francisco, CA, USA, Dec. 3–9, 2023). Ed. by Satish Chandra, Kelly Blincoe, and Paolo Tonella. New York, NY, USA: ACM, 2023, pp. 450–462. ISBN: 979-8-4007-0327-0. DOI: 10.1145/3611643.3616325.

[53] Kaifeng Huang, Bihuan Chen, Congying Xu, Ying Wang, Bowen Shi, Xin Peng, Yijian Wu, and Yang Liu. "Characterizing usages, updates and risks of third-party libraries in Java projects". In: *Empirical Software Engineering* 27.4, 90 (July 2022): *Software Maintenance and Evolution (ICSME)*. Ed. by Zhenchang Xing and Kelly Blincoe, 41 pp. ISSN: 1573-7616. DOI: 10.1007/S10664-022-10131-8.

[54] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. 1st ed. Boston, MA, USA: Addison-Wesley, 2010. ISBN: 978-0-321-60191-9.

[55] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. "Testing Idempotence for Infrastructure as Code". In: *Middleware 2013 – ACM/IFIP/USENIX 14th International Middleware Conference, Proceedings*. Middleware 2013 (Beijing, China, Dec. 9, 2013–Dec. 13, 2023). Ed. by David M. Eyers and Karsten Schwan. Berlin, Germany: Springer, 2013, pp. 368–388. ISBN: 978-3-642-45064-8. DOI: 10.1007/978-3-642-45065-5_19.

[56] Md Hasan Ibrahim, Mohammed Sayagh, and Ahmed E. Hassan. "Too many images on DockerHub! How different are images for the same system?" In: *Empirical Software Engineering* 25.5 (Sept. 2020). Ed. by Nachiappan Nagappan, pp. 4250–4281. ISSN: 1573-7616. DOI: 10.1007/S10664-020-09873-0.

[57] Katsuhiko Ikeshita, Fuyuki Ishikawa, and Shinichi Honiden. "Test Suite Reduction in Idempotence Testing of Infrastructure as Code". In: *Tests and Proofs – 11th International Conference, Proceedings*. TAP 2017 (Marburg, Germany, July 19–20, 2017). Ed. by Sebastian Gabmeyer and Einar Broch Johnsen. Lecture Notes in Computer Science 10375. Berlin, Germany: Springer, 2017, pp. 98–115. ISBN: 978-3-319-61466-3. DOI: 10.1007/978-3-319-61467-0_6.

[58] Md Rakibul Islam, Minhaz F. Zibran, and Aayush Nagpal. "Security Vulnerabilities in Categories of Clones and Non-Cloned Code: An Empirical Study". In: *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM 2017 (Toronto, ON, Canada, Nov. 9–10, 2017). Ed. by Ayse Bener, Burak

Turhan, and Stefan Biffl. Los Alamitos, CA, USA: IEEE, 2017, pp. 20–29. ISBN: 978-1-5090-4039-1. DOI: 10.1109/ESEM.2017.9.

[59] Ling Jiang, Hengchen Yuan, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. "Third-Party Library Dependency for Large-Scale SCA in the C/C++ Ecosystem: How Far Are We?" In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2023 (Seattle, WA, USA, July 17–21, 2023). Ed. by René Just and Gordon Fraser. New York, NY, USA: ACM, 2023, pp. 1383–1395. ISBN: 979-8-4007-0221-1. DOI: 10.1145/3597926.3598143.

[60] Yujuan Jiang and Bram Adams. "Co-evolution of Infrastructure and Source Code - An Empirical Study". In: *Proceedings of the 12th IEEE/ACM Working Conference on Mining Software Repositories*. MSR 2015 (Florence, Italy, May 16–17, 2015). Ed. by Massimiliano Di Penta, Martin Pinzger, and Romain Robbes. Los Alamitos, CA, USA: IEEE, 2015, pp. 45–55. ISBN: 978-0-7695-5594-2. DOI: 10.1109/MSR.2015.12.

[61] joern.io. *Joern: The Bug Hunter's Workbench*. Version 2.0. Jan. 2024. URL: https://github.com/joernio/joern (visited on Jan. 2024).

[62] Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. "Exploring and Enforcing Security Guarantees via Program Dependence Graphs". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2015 (Portland, OR, USA, June 15–17, 2015). Ed. by David Grove and Stephen M. Blackburn. New York, NY, USA: ACM, 2015, pp. 291–302. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737957.

[63] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. "Structure and evolution of package dependency networks". In: *Proceedings of the 14th IEEE/ACM International Conference on Mining Software Repositories*. MSR 2017 (Buenos Aires, Argentina, May 20–21, 2017). Ed. by Jesús M. González-Barahona, Abram Hindle, and Lin Tan. Los Alamitos, CA, USA: IEEE, 2017, pp. 102–112. ISBN: 978-1-5386-1544-7. DOI: 10.1109/MSR.2017.55.

[64] John P. Klein and Melvin L. Moeschberger. *Survival Analysis: Techniques for Censored and Truncated Data*. 2nd ed. Berlin, Germany: Springer, 2003. ISBN: 978-0-387-95399-1.

[65] Shoma Kokuryo, Masanari Kondo, and Osamu Mizuno. "An Empirical Study of Utilization of Imperative Modules in Ansible". In: *Proceedings of the 20th IEEE International Conference on Software Quality, Reliability and Security*. QRS 2020 (Macao, China, Dec. 11–14, 2020). Ed. by Shaoying Liu, Du Zhang, Vytautas Bučinskas, W.K. Chan, Mei Nagappan, and Christof Budnik. Los Alamitos, CA, USA: IEEE, 2020, pp. 442–449. ISBN: 978-1-7281-8913-0. DOI: 10.1109/QRS51102.2020.00063.

[66] Christoph Krieger, Uwe Breitenbücher, Kálmán Képes, and Frank Leymann. "An Approach to Automatically Check the Compliance of Declarative Deployment Models". In: *Papers from the 12th Advanced Summer School on Service-Oriented Computing*. SummerSOC 2018 (Hersonissos, Crete, Greece, June 24–29, 2018). Ed. by Johanna Barzen,

Rania Khalaf, Frank Leymann, and Bernhard Mitschang. IBM Research Division. 2018, pp. 76–89. IBM Research Reports: RC25681.

[67] Indika Kumara, Martín Garriga, Angel Urbano Romeu, Dario Di Nucci, Fabio Palomba, Damian Andrew Tamburri, and Willem-Jan van den Heuvel. "The do's and don'ts of infrastructure code: A systematic gray literature review". In: *Information and Software Technology* 137, 106593 (Sept. 2021). Ed. by Günther Ruhe, 20 pp. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2021.106593.

[68] Indika Kumara, Zoe Vasileiou, Georgios Meditskos, Damian Andrew Tamburri, Willem-Jan van den Heuvel, Anastasios Karakostas, Stefanos Vrochidis, and Ioannis Kompatsiaris. "Towards Semantic Detection of Smells in Cloud Infrastructure Code". In: *Proceedings of the 10th International Conference on Web Intelligence, Mining and Semantics*. WIMS 2020 (Biarritz, France, June 30–July 3, 2020). Ed. by Richard Chbeir, Yannis Manolopoulos, Rajendra Akerkar, and Jolanta Mizera-Pietraszko. New York, NY, USA: ACM, 2020, pp. 63–67. ISBN: 978-1-4503-7542-9. DOI: 10.1145/3405962.3405979.

[69] Julien Lepiller, Ruzica Piskac, Martin Schäf, and Mark Santolucito. "Analyzing Infrastructure as Code to Prevent Intra-update Sniping Vulnerabilities". In: *Tools and Algorithms for the Construction and Analysis of Systems – 27th International Conference, Proceedings, Part II*. TACAS 2021 (Luxembourg City, Luxembourg, Mar. 27–Apr. 1, 2021). Ed. by Jan Friso Groote and Kim Guldstrand Larsen. Lecture Notes in Computer Science 12652. Berlin, Germany: Springer, 2021, pp. 105–123. ISBN: 978-3-030-72012-4. DOI: 10.1007/978-3-030-72013-1_6.

[70] Changyuan Lin, Sarah Nadi, and Hamzeh Khazaei. "A Large-scale Data Set and an Empirical Study of Docker Images Hosted on Docker Hub". In: *Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution*. ICSME 2020 (Virtual Event, Adelaide, Australia, Sept. 28–Oct. 2, 2020). Ed. by Christoph Treude, Hongyu Zhang, Kelly Blincoe, and Zhenchang Xing. Los Alamitos, CA, USA: IEEE, 2020, pp. 371–381. ISBN: 978-1-7281-5619-4. DOI: 10.1109/ICSME46990.2020.00043.

[71] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. "GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis". In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD 2006 (Philadelphia, PA, USA, Aug. 20–23, 2006). Ed. by Tina Eliassi-Rad, Lyle H. Ungar, Mark Craven, and Dimitrios Gunopulos. New York, NY, USA: ACM, 2006, pp. 872–881. ISBN: 1-59593-339-5. DOI: 10.1145/1150402.1150522.

[72] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. "Demystifying the Vulnerability Propagation and Its Evolution via Dependency Trees in the NPM Ecosystem". In: *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering*. ICSE 2022 (Pittsburgh, PA, USA, May 25–27, 2022). Ed. by Matthew B. Dwyer, Daniela Damian, and Andreas Zeller. New York, NY, USA: ACM, 2022, pp. 672–684. ISBN: 978-1-4503-9221-1. DOI: 10.1145/3510003.3510142.

[73] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, and Jan Vitek. "DéjàVu: a map of code duplicates on GitHub". In: *Proceedings of the ACM on Programming Languages* 1, 84 (OOPSLA Oct. 2017). Ed. by Philip Wadler, 28 pp. ISSN: 2475-1421. DOI: 10.1145/3133908.

[74] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. 1st ed. Cambridge, UK: Cambridge University Press, 2008. ISBN: 978-0-511-41405-3. DOI: 10.1017/CBO9780511809071.

[75] Gary McGraw and Bruce Potter. "Software Security Testing". In: *IEEE Security & Privacy* 2.5 (Sept. 2004). Ed. by Gary McGraw, pp. 81–85. ISSN: 1540-7993. DOI: 10.1109/MSP.2004.84.

[76] Bas Meijer, Lorin Hochstein, and René Moser. *Ansible: Up and Running*. 3rd ed. Sebastopol, CA, USA: O'Reilly, 2022. ISBN: 978-1-098-10915-8.

[77] Manishankar Mondal, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. "An empirical study on bug propagation through code cloning". In: *Journal of Systems and Software* 158, 110407 (Dec. 2019). Ed. by Paris Avgeriou and David Shepherd, 18 pp. ISSN: 0164-1212. DOI: 10.1016/J.JSS.2019.110407.

[78] Kief Morris. *Infrastructure as Code: Managing Servers in the Cloud*. 1st ed. Sebastopol, CA, USA: O'Reilly, 2016. ISBN: 978-1-491-92435-8.

[79] Hoan Anh Nguyen, Tien N. Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. "Graph-Based Mining of in-the-Wild, Fine-Grained, Semantic Code Change Patterns". In: *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering*. ICSE 2019 (Montreal, Canada, May 25–31, 2019). Ed. by Joanne M. Atlee, Tevfik Bultan, and Jon Whittle. Los Alamitos, CA, USA: IEEE, 2019, pp. 819–830. ISBN: 978-1-7281-0869-8. DOI: 10.1109/ICSE.2019.00089.

[80] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. "Graph-Based Mining of Multiple Object Usage Patterns". In: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC/FSE 2009 (Amsterdam, The Netherlands, Aug. 24–28, 2009). Ed. by Hans van Vliet and Valérie Issarny. New York, NY, USA: ACM, 2009, pp. 383–392. ISBN: 978-1-60558-001-2. DOI: 10.1145/1595696.1595767.

[81] Evangelos Ntentos, Uwe Zdun, Ghareeb Falazi, Uwe Breitenbücher, and Frank Leymann. "Assessing Architecture Conformance to Security-Related Practices in Infrastructure as Code Based Deployments". In: *Proceedings of the 2022 IEEE International Conference on Services Computing*. SCC 2022 (Barcelona, Spain, July 11–15, 2022). Ed. by Claudio Agostino Ardagna et al. Los Alamitos, CA, USA: IEEE, 2022, pp. 123–133. ISBN: 978-1-6654-8146-5. DOI: 10.1109/SCC55611.2022.00029.

[82] Evangelos Ntentos, Uwe Zdun, Ghareeb Falazi, Uwe Breitenbücher, and Frank Leymann. "Detecting and Resolving Coupling-Related Infrastructure as Code Based Architecture Smells in Microservice Deployments". In: *Proceedings of the 16th IEEE International Conference on*

Cloud Computing. CLOUD 2023 (Chicago, IL, USA, July 2–8, 2023). Ed. by Claudio Ardagna et al. Los Alamitos, CA, USA: IEEE, 2023, pp. 201–211. ISBN: 979-8-3503-0481-7. DOI: 10.1109/CLOUD60044.2023.00031.

[83] Evangelos Ntentos, Uwe Zdun, Jacopo Soldani, and Antonio Brogi. "Assessing Architecture Conformance to Coupling-Related Infrastructure-as-Code Best Practices: Metrics and Case Studies". In: *Software Architecture – 16th European Conference, Proceedings*. ECSA 2022 (Prague, Czech Republic, Sept. 19–23, 2022). Ed. by Ilias Gerostathopoulos, Grace A. Lewis, Thaís Vasconcelos Batista, and Tomás Bures. Lecture Notes in Computer Science 13444. Berlin, Germany: Springer, 2022, pp. 101–116. ISBN: 978-3-031-16696-9. DOI: 10.1007/978-3-031-16697-6_7.

[84] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks". In: *Detection of Intrusions and Malware, and Vulnerability Assessment – 17th International Conference, Proceedings*. DIMVA 2020 (Lisbon, Portugal, June 24–26, 2020). Ed. by Clémentine Maurice, Leyla Bilge, Gianluca Stringhini, and Nuno Neves. Lecture Notes in Computer Science 12223. Berlin, Germany: Springer, 2020, pp. 23–43. ISBN: 978-3-030-52683-2. DOI: 10.1007/978-3-030-52683-2_2.

[85] Hassan Onsori Delicheh, Alexandre Decan, and Tom Mens. "Quantifying Security Issues in Reusable JavaScript Actions in GitHub Workflows". In: *Proceedings of the 21st International Conference on Mining Software Repositories*. MSR 2024 (Lisbon, Portugal, Apr. 15–May 16, 2024). Ed. by Diomidis Spinellis, Alberto Bacchelli, and Eleni Constantinou. New York, NY, USA: ACM, 2024, pp. 692–703. ISBN: 979-8-4007-0587-8. DOI: 10.1145/3643991.3644899.

[86] Ruben Opdebeeck and Coen De Roover. *Ansible Is Turing Complete*. Presentation abstract at the 2nd Workshop on Configuration Languages, CONFLANG'23. 2023. URL: https://researchportal.vub.be/en/publications/ansible-is-turing-complete-presentation-abstract.

[87] Ruben Opdebeeck, Johan Fabry, Tim Molderez, Jonas De Bleser, and Coen De Roover. "Mining for Graph-Based Library Usage Patterns in COBOL Systems". In: *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering*. SANER 2021 (Virtual Event, Honolulu, HI, USA, Mar. 9–12, 2021). Ed. by Rick Kazman, Yuanfang Cai, and Marouane Kessentini. Los Alamitos, CA, USA: IEEE, 2021, pp. 595–599. ISBN: 978-1-7281-9630-5. DOI: 10.1109/SANER50967.2021.00072.

[88] Ruben Opdebeeck, Jonas Lesy, Ahmed Zerouali, and Coen De Roover. "The Docker Hub Image Inheritance Network: Construction and Empirical Insights". In: *Proceedings of the 23rd IEEE International Working Conference on Source Code Analysis and Manipulation*. SCAM 2023 (Bogotá, Colombia, Oct. 2–3, 2023). Ed. by Leon Moonen, Christian D. Newman, and Alessandra Gorla. Los Alamitos, CA, USA: IEEE, 2023, pp. 198–208. ISBN: 979-8-3503-0506-7. DOI: 10.1109/SCAM59687.2023.00029.

[89] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. "Andromeda: A Dataset of Ansible Galaxy Roles and Their Evolution". In: *Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories*. MSR 2021 (Virtual Event, Madrid, Spain, May 17–19, 2021). Ed. by Gregorio Robles, Kelly Blincoe, and Meiyappan Nagappan. Los Alamitos, CA, USA: IEEE, 2021, pp. 580–584. ISBN: 978-1-7281-8710-5. DOI: 10.1109/MSR52588.2021.00078.

[90] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. "Control and Data Flow in Security Smell Detection for Infrastructure as Code: Is It Worth the Effort?" In: *Proceedings of the 20th IEEE/ACM International Conference on Mining Software Repositories*. MSR 2023 (Melbourne, Australia, May 15–16, 2023). Ed. by Emad Shihab, Patanamon Thongtanunam, and Bogdan Vasilescu. Los Alamitos, CA, USA: IEEE, 2023, pp. 534–545. ISBN: 979-8-3503-1184-6. DOI: 10.1109/MSR59073.2023.00079.

[91] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. "Infrastructure-as-Code Ecosystems". In: *Software Ecosystems: Tooling and Analytics*. Ed. by Tom Mens, Coen De Roover, and Anthony Cleve. Berlin, Germany: Springer, 2023, pp. 215–245. ISBN: 978-3-031-36060-2. DOI: 10.1007/978-3-031-36060-2_9.

[92] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. "Smelly Variables in Ansible Infrastructure Code: Detection, Prevalence, and Lifetime". In: *Proceedings of the 19th IEEE/ACM International Conference on Mining Software Repositories*. MSR 2022 (Pittsburgh, PA, USA, May 23–24, 2022). Ed. by David Lo, Shane McIntosh, and Nicole Novielli. New York, NY, USA: ACM, 2022, pp. 61–72. ISBN: 978-1-4503-9303-4. DOI: 10.1145/3524842.3527964.

[93] Ruben Opdebeeck, Ahmed Zerouali, Camilo Velázquez-Rodríguez, and Coen De Roover. "Does Infrastructure as Code Adhere to Semantic Versioning? An Analysis of Ansible Role Evolution". In: *Proceedings of the 20th IEEE International Working Conference on Source Code Analysis and Manipulation*. SCAM 2020 (Virtual Event, Adelaide, Australia, Sept. 27–28, 2020). Ed. by Foutse Khomh, Cristina Cifuentes, and Nikolaos Tsantalis. Los Alamitos, CA, USA: IEEE, 2020, pp. 238–248. ISBN: 978-1-7281-9248-2. DOI: 10.1109/SCAM51674.2020.00032.

[94] Ruben Opdebeeck, Ahmed Zerouali, Camilo Velázquez-Rodríguez, and Coen De Roover. "On the practice of semantic versioning for Ansible Galaxy roles: An empirical study and a change classification model". In: *Journal of Systems and Software* 182, 111059 (Dec. 2021): *Special Section on Source Code Analysis and Manipulation*. Ed. by Wing-Kwong Chan, Tsantalis Nikolaos, and Cristina Cifuentes, 21 pp. ISSN: 0164-1212. DOI: 10.1016/j.jss.2021.111059.

[95] Karl J. Ottenstein and Linda M. Ottenstein. "The Program Dependence Graph in a Software Development Environment". In: *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. SDE 1984 (Pittsburgh, PA, USA, Apr. 23–25, 1984). Ed. by William E. Riddle and Peter B. Henderson.

New York, NY, USA: ACM, 1984, pp. 177–184. ISBN: 0-89791-131-8. DOI: 10.1145/800020.808263.

[96] Laure Philips, Joeri De Koster, Wolfgang De Meuter, and Coen De Roover. "Dependence-Driven Delimited CPS Transformation for JavaScript". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2016 (Amsterdam, The Netherlands, Oct. 31–Nov. 1, 2016). Ed. by Bernd Fischer and Ina Schaefer. New York, NY, USA: ACM, 2016, pp. 59–69. ISBN: 978-1-4503-4446-3. DOI: 10.1145/2993236.2993243.

[97] Laure Philips, Joeri De Koster, Wolfgang De Meuter, and Coen De Roover. "Search-based Tier Assignment for Optimising Offline Availability in Multi-tier Web Applications". In: *The Art, Science, and Engineering of Programming* 2.2, 3 (Dec. 2018). Ed. by Cristina V. Lopes and Guido Salvaneschi, 29 pp. ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2018/2/3.

[98] Giovanni Quattrocchi and Damian Andrew Tamburri. "Predictive maintenance of infrastructure code using "fluid" datasets: An exploratory study on Ansible defect proneness". In: *Journal of Software: Evolution and Process* 34.11, e2480 (Nov. 2022): *Special Issue: Automatic Software Testing from the Trenches*. Ed. by Breno Miranda, Javier Tuya, and Alejandra Garrido, 26 pp. ISSN: 2047-7473. DOI: 10.1002/SMR.2480.

[99] Steven Raemaekers, Arie van Deursen, and Joost Visser. "Semantic Versioning versus Breaking Changes: A Study of the Maven Repository". In: *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation*. SCAM 2014 (Victoria, BC, Canada, Sept. 28–29, 2014). Ed. by Dawn Lawrie, Abram Hindle, and Rocco Oliveto. Los Alamitos, CA, USA: IEEE, 2014, pp. 215–224. ISBN: 978-0-7695-5304-7. DOI: 10.1109/SCAM.2014.30.

[100] Akond Rahman, Farhat Lamia Barsha, and Patrick Morrison. "Shhh!: 12 Practices for Secret Management in Infrastructure as Code". In: *Proceedings of the 2021 IEEE Secure Development Conference*. SecDev 2021 (Virtual Event, Atlanta, GA, USA, Oct. 8–10, 2021). Ed. by Brendan D. Saltaformaggio, Jason Li, Limin Jia, and Frank Piessens. Los Alamitos, CA, USA: IEEE, 2021, pp. 56–62. ISBN: 978-1-6654-3170-5. DOI: 10.1109/SecDev51306.2021.00024.

[101] Akond Rahman, Dibyendu Brinto Bose, Yue Zhang, and Rahul Pandita. "An empirical study of task infections in Ansible scripts". In: *Empirical Software Engineering* 29.1, 34 (Jan. 2024). Ed. by Mika Mäntylä, 44 pp. ISSN: 1573-7616. DOI: 10.1007/S10664-023-10432-6.

[102] Akond Rahman, Effat Farhana, Chris Parnin, and Laurie Williams. "Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts". In: *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering*. ICSE 2020 (Virtual Event, Seoul, South Korea, June 27–July 19, 2020). Ed. by Gregg Rothermel and Doo-Hwan Bae. New York, NY, USA: ACM, 2020, pp. 752–764. ISBN: 978-1-4503-7121-6. DOI: 10.1145/3377811.3380409.

[103] Akond Rahman, Effat Farhana, and Laurie Williams. "The 'as code' activities: development anti-patterns for infrastructure as code". In: *Empirical Software Engineering* 25.5 (Sept. 2020). Ed. by Daniel Méndez, pp. 3430–3467. ISSN: 1573-7616. DOI: 10.1007/S10664-020-09841-8.

[104] Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie Williams. "A systematic mapping study of infrastructure as code research". In: *Information and Software Technology* 108 (Apr. 2019). Ed. by Günther Ruhe, pp. 65–77. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2018.12.004.

[105] Akond Rahman and Chris Parnin. "Detecting and Characterizing Propagation of Security Weaknesses in Puppet-Based Infrastructure Management". In: *IEEE Transactions on Software Engineering* 49.6 (June 2023). Ed. by Mehdi Mirakhorli, pp. 3536–3553. ISSN: 0098-5589. DOI: 10.1109/TSE.2023.3265962.

[106] Akond Rahman, Chris Parnin, and Laurie Williams. "The Seven Sins: Security Smells in Infrastructure as Code Scripts". In: *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering*. ICSE 2019 (Montreal, Canada, May 25–31, 2019). Ed. by Joanne M. Atlee, Tevfik Bultan, and Jon Whittle. Los Alamitos, CA, USA: IEEE, 2019, pp. 164–175. ISBN: 978-1-7281-0869-8. DOI: 10.1109/ICSE.2019.00033.

[107] Akond Rahman, Asif Partho, Patrick Morrison, and Laurie Williams. "What questions do programmers ask about configuration as code?" In: *Proceedings of the 4th International Workshop on Rapid Continuous Software Engineering*. RCoSE 2018 (Gothenburg, Sweden, May 29, 2018). Ed. by Jan Bosch, Brian Fitzgerald, Michael Goedicke, Helena Holmström Olsson, Marco Konersmann, and Stephan Krusche. New York, NY, USA: ACM, 2018, pp. 16–22. ISBN: 978-1-4503-5745-6. DOI: 10.1145/3194760.3194769.

[108] Akond Rahman, Md Rayhanur Rahman, Chris Parnin, and Laurie Williams. "Security Smells in Ansible and Chef Scripts: A Replication Study". In: *ACM Transactions on Software Engineering and Methodology* 30.1, 3 (Jan. 2021). Ed. by Mauro Pezzè, 31 pp. ISSN: 1049-331X. DOI: 10.1145/3408897.

[109] Akond Rahman, Shazibul Islam Shamim, Dibyendu Brinto Bose, and Rahul Pandita. "Security Misconfigurations in Open Source Kubernetes Manifests: An Empirical Study". In: *ACM Transactions on Software Engineering and Methodology* 32.4, 99 (July 2023). Ed. by Mauro Pezzè, 36 pp. ISSN: 1049-331X. DOI: 10.1145/3579639.

[110] Akond Rahman and Tushar Sharma. "Lessons from Research to Practice on Writing Better Quality Puppet Scripts". In: *Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering*. SANER 2022 (Virtual Event, Honolulu, HI, USA, Mar. 15–18, 2022). Ed. by Alexander Chatzigeorgiou, Dan Hao, and Abram Hindle. Los Alamitos, CA, USA: IEEE, 2022, pp. 63–67. ISBN: 978-1-6654-3786-8. DOI: 10.1109/SANER53432.2022.00019.

[111] Akond Rahman and Laurie Williams. "Characterizing Defective Configuration Scripts Used for Continuous Deployment". In: *Proceedings of the*

11th IEEE International Conference on Software Testing, Verification and Validation. ICST 2018 (Västerås, Sweden, Apr. 9–13, 2018). Ed. by Hans Hansson, Robert Feldt, and Shin Yoo. Los Alamitos, CA, USA: IEEE, 2018, pp. 34–45. ISBN: 978-1-5386-5012-7. DOI: 10.1109/ICST.2018.00014.

[112] Akond Rahman and Laurie Williams. "Source code properties of defective infrastructure as code scripts". In: *Information and Software Technology* 112 (Aug. 2019). Ed. by Günther Ruhe, pp. 148–163. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2019.04.013.

[113] Sofia Reis, Rui Abreu, Marcelo d'Amorim, and Daniel Fortunato. "Leveraging Practitioners' Feedback to Improve a Security Linter". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2022 (Rochester, MI, USA, Oct. 10–14, 2022). Ed. by Marouane Kessentini, Julia Rubin, and Shahar Maoz. New York, NY, USA: ACM, 2022, 66, 12 pp. ISBN: 978-1-4503-9475-8. DOI: 10.1145/3551349.3560419.

[114] Benjamin Rombaut, Filipe Roseiro Côgo, Bram Adams, and Ahmed E. Hassan. "There's no Such Thing as a Free Lunch: Lessons Learned from Exploring the Overhead Introduced by the Greenkeeper Dependency Bot in Npm". In: *ACM Transactions on Software Engineering and Methodology* 32.1, 11 (Jan. 2023). Ed. by Mauro Pezzè, 40 pp. ISSN: 1049-331X. DOI: 10.1145/3522587.

[115] Nuno Saavedra and João F. Ferreira. "GLITCH: Automated Polyglot Security Smell Detection in Infrastructure as Code". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2022 (Rochester, MI, USA, Oct. 10–14, 2022). Ed. by Marouane Kessentini, Julia Rubin, and Shahar Maoz. New York, NY, USA: ACM, 2022, 47, 12 pp. ISBN: 978-1-4503-9475-8. DOI: 10.1145/3551349.3556945.

[116] Nuno Saavedra, João Gonçalves, Miguel Henriques, João F. Ferreira, and Alexandra Mendes. "Polyglot Code Smell Detection for Infrastructure as Code with GLITCH". In: *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2023 (Echternach, Luxembourg, Sept. 11–15, 2023). Ed. by Tegawendé F. Bissyandé, Jacques Klein, Christian Bird, and Federica Sarro. Los Alamitos, CA, USA: IEEE, 2023, pp. 2042–2045. ISBN: 979-8-3503-2996-4. DOI: 10.1109/ASE56229.2023.00162.

[117] Julian Schwarz, Andreas Steffens, and Horst Lichter. "Code Smells in Infrastructure as Code". In: *Proceedings of the 11th International Conference on the Quality of Information and Communications Technology*. QUATIC 2018 (Coimbra, Portugal, Sept. 4–7, 2018). Ed. by Antonia Bertolino, Vasco Amaral, Paulo Rupino, and Marco Vieira. Los Alamitos, CA, USA: IEEE, 2018, pp. 220–228. ISBN: 978-1-5386-5841-3. DOI: 10.1109/QUATIC.2018.00040.

[118] Gehan M. K. Selim, Liliane Barbour, Weiyi Shang, Bram Adams, Ahmed E. Hassan, and Ying Zou. "Studying the Impact of Clones on Software Defects". In: *Proceedings of the 17th Working Conference on Reverse Engineering*. WCRE 2010 (Beverly, MA, USA, Oct. 13–16, 2010). Ed.

by Giuliano Antoniol, Martin Pinzger, and Elliot J. Chikofsky. Los Alamitos, CA, USA: IEEE, 2010, pp. 13–21. ISBN: 978-0-7695-4123-5. DOI: 10.1109/WCRE.2010.11.

[119] Rian Shambaugh, Aaron Weiss, and Arjun Guha. "Rehearsal: A Configuration Verification Tool for Puppet". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2016 (Santa Barbara, CA, USA, June 13–17, 2016). Ed. by Chandra Krintz and Emery D. Berger. New York, NY, USA: ACM, 2016, pp. 416–430. ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908083.

[120] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. "Does Your Configuration Code Smell?" In: *Proceedings of the 13th International Conference on Mining Software Repositories*. MSR 2016 (Austin, TX, USA, May 14–15, 2016). Ed. by Miryung Kim, Romain Robbes, and Christian Bird. New York, NY, USA: ACM, 2016, pp. 189–200. ISBN: 978-1-4503-4186-8. DOI: 10.1145/2901739.2901761.

[121] Tushar Sharma and Diomidis Spinellis. "A survey on software smells". In: *Journal of Systems and Software* 138 (Apr. 2018). Ed. by Paris Avgeriou and David Shepherd, pp. 158–173. ISSN: 0164-1212. DOI: 10.1016/j.jss.2017.12.034.

[122] Ryo Shimizu and Hideyuki Kanuka. "Test-Based Least Privilege Discovery on Cloud Infrastructure as Code". In: *Proceedings of the 12th IEEE International Conference on Cloud Computing Technology and Science*. CloudCom 2020 (Bangkok, Thailand, Dec. 14–17, 2020). Ed. by Hyong Kim, Akkarit Sangpetch, Orathai Sangpetch, Kanat Tangwongsan, and Kyriaki Levanti. Los Alamitos, CA, USA: IEEE, 2020, pp. 1–8. ISBN: 978-1-6654-0388-7. DOI: 10.1109/CloudCom49646.2020.00007.

[123] Ryo Shimizu, Yuna Nunomura, and Hideyuki Kanuka. "Test-suite-guided discovery of least privilege for cloud infrastructure as code". In: *Automated Software Engineering* 31.1, 25 (May 2024). Ed. by Tim Menzies, 39 pp. ISSN: 1573-7535. DOI: 10.1007/S10515-024-00420-5.

[124] Charlie Soh, Hee Beng Kuan Tan, Yauhen Leanidavich Arnatovich, Annamalai Narayanan, and Lipo Wang. "LibSift: Automated Detection of Third-Party Libraries in Android Applications". In: *Proceedings of the 23rd Asia-Pacific Software Engineering Conference*. APSEC 2016 (Hamilton, New Zealand, Dec. 6–9, 2016). Ed. by Alex Potanin, Gail C. Murphy, Steve Reeves, and Jens Dietrich. Los Alamitos, CA, USA: IEEE, 2016, pp. 41–48. ISBN: 978-1-5090-5575-3. DOI: 10.1109/APSEC.2016.017.

[125] Daniel Sokolowski and Guido Salvaneschi. "Towards Reliable Infrastructure as Code". In: *Companion Proceedings of the 20th International Conference on Software Architecture*. 2nd International Workshop on the Foundations of Infrastructure Specification and Testing (FIST 2023) (L'Aquila, Italy, Mar. 14–17, 2023). Ed. by Luciano Baresi, Giovanni Quattrocchi, and Damian Andrew Tamburri. Los Alamitos, CA, USA: IEEE, 2023, pp. 318–321. ISBN: 978-1-6654-6459-8. DOI: 10.1109/ICSA-C57050.2023.00072.

[126] Daniel Sokolowski, David Spielmann, and Guido Salvaneschi. "Automated Infrastructure as Code Program Testing". In: *IEEE Transactions on*

Software Engineering 50.6 (June 2024). Ed. by Per Runeson, pp. 1585–1599. ISSN: 0098-5589. DOI: 10.1109/TSE.2024.3393070.

[127] Thodoris Sotiropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. "Practical Fault Detection in Puppet Programs". In: *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering*. ICSE 2020 (Virtual Event, Seoul, South Korea, June 27–July 19, 2020). Ed. by Gregg Rothermel and Doo-Hwan Bae. New York, NY, USA: ACM, 2020, pp. 26–37. ISBN: 978-1-4503-7121-6. DOI: 10.1145/3377811.3380384.

[128] David Spielmann, Daniel Sokolowski, and Guido Salvaneschi. "Extensible Testing for Infrastructure as Code". In: *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. SPLASH 2023 (Cascais, Portugal, Oct. 22–27, 2023). Ed. by Vasco Thudichum Vasconcelos. New York, NY, USA: ACM, 2023, pp. 58–60. ISBN: 979-8-4007-0384-3. DOI: 10.1145/3618305.3623607.

[129] StackExchange, Inc. *2022 Annual StackOverflow Developer Survey*. 2022. URL: https://survey.stackoverflow.co/2022/#section-most-popular-technologies-other-tools (visited on Jan. 10, 2023).

[130] StackExchange, Inc. *2023 Annual StackOverflow Developer Survey*. 2023. URL: https://survey.stackoverflow.co/2023/#section-most-popular-technologies-other-tools (visited on Mar. 20, 2023).

[131] Quentin Stiévenart, Dave Binkley, and Coen De Roover. "QSES: Quasi-Static Executable Slices". In: *Proceedings of the 21th IEEE International Working Conference on Source Code Analysis and Manipulation*. SCAM 2021 (Virtual, Luxembourg City, Luxembourg, Sept. 27–Oct. 1, 2021). Ed. by Alexander Serebrenik, Venera Arnaoudova, and Ben Hermann. Los Alamitos, CA, USA: IEEE, 2021, pp. 209–213. ISBN: 978-1-6654-4897-0. DOI: 10.1109/SCAM52516.2021.00033.

[132] Quentin Stiévenart, Dave Binkley, and Coen De Roover. "Static Stack-Preserving Intra-Procedural Slicing of WebAssembly Binaries". In: *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering*. ICSE 2022 (Pittsburgh, PA, USA, May 25–27, 2022). Ed. by Matthew B. Dwyer, Daniela Damian, and Andreas Zeller. New York, NY, USA: ACM, 2022, pp. 2031–2042. ISBN: 978-1-4503-9221-1. DOI: 10.1145/3510003.3510070.

[133] Wei Tang, Du Chen, and Ping Luo. "BCFinder: A Lightweight and Platform-Independent Tool to Find Third-Party Components in Binaries". In: *Proceedings of the 25th Asia-Pacific Software Engineering Conference*. APSEC 2018 (Nara, Japan, Dec. 4–7, 2018). Ed. by Katsuhisa Maruyama, Naoyasu Ubayashi, Hironori Washizaki, and Hongyu Zhang. Los Alamitos, CA, USA: IEEE, 2018, pp. 288–297. ISBN: 978-1-7281-1970-0. DOI: 10.1109/APSEC.2018.00043.

[134] Wei Tang, Zhengzi Xu, Chengwei Liu, Jiahui Wu, Shouguo Yang, Yi Li, Ping Luo, and Yang Liu. "Towards Understanding Third-party Library Dependency in C/C++ Ecosystem". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2022 (Rochester, MI, USA, Oct. 10–14, 2022). Ed. by Marouane Kessentini,

Julia Rubin, and Shahar Maoz. New York, NY, USA: ACM, 2022, 106, 12 pp. ISBN: 978-1-4503-9475-8. DOI: 10.1145/3551349.3560432.

[135] Eduard van der Bent, Jurriaan Hage, Joost Visser, and Georgios Gousios. "How Good is Your Puppet? An Empirically Defined and Validated Quality Model for Puppet". In: *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering*. SANER 2018 (Campobasso, Italy, Mar. 20–23, 2018). Ed. by Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd. Los Alamitos, CA, USA: IEEE, 2018, pp. 164–174. ISBN: 978-1-5386-4969-5. DOI: 10.1109/SANER.2018.8330206.

[136] Rosemary Wang. *Infrastructure as Code, Patterns and Practices*. 1st ed. Shelter Island, NY, USA: Manning, 2022. ISBN: 978-1-617-29829-5.

[137] Ying Wang, Ming Wen, Yepang Liu, Yibo Wang, Zhenming Li, Chao Wang, Hai Yu, Shing-Chi Cheung, Chang Xu, and Zhiliang Zhu. "Watchman: monitoring dependency conflicts for Python library ecosystem". In: *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering*. ICSE 2020 (Virtual Event, Seoul, South Korea, June 27–July 19, 2020). Ed. by Gregg Rothermel and Doo-Hwan Bae. New York, NY, USA: ACM, 2020, pp. 125–135. ISBN: 978-1-4503-7121-6. DOI: 10.1145/3377811.3380426.

[138] Aaron Weiss, Arjun Guha, and Yuriy Brun. "Tortoise: Interactive system configuration repair". In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. ASE 2017 (Urbana-Champaign, IL, USA, Oct. 30–Nov. 3, 2017). Ed. by Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen. Los Alamitos, CA, USA: IEEE, 2017, pp. 625–636. ISBN: 978-1-5386-2684-9. DOI: 10.1109/ASE.2017.8115673.

[139] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. "A look at the dynamics of the JavaScript package ecosystem". In: *Proceedings of the 13th International Conference on Mining Software Repositories*. MSR 2016 (Austin, TX, USA, May 14–15, 2016). Ed. by Miryung Kim, Romain Robbes, and Christian Bird. New York, NY, USA: ACM, 2016, pp. 351–361. ISBN: 978-1-4503-4186-8. DOI: 10.1145/2901739.2901743.

[140] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. 1st ed. Berlin, Germany: Springer, 2012. ISBN: 978-3-642-29044-2. DOI: 10.1007/978-3-642-29044-2.

[141] Seunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh. "Centris: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse". In: *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering*. ICSE 2021 (Virtual Event, Madrid, Spain, May 25–28, 2021). Ed. by Natalia Juristo, Arie van Deursen, and Tao Xie. Los Alamitos, CA, USA: IEEE, 2021, pp. 860–872. ISBN: 978-1-6654-4831-4. DOI: 10.1109/ICSE43902.2021.00083.

[142] Di Wu, Lin Chen, Yuming Zhou, and Baowen Xu. "How do developers use C++ libraries? An empirical study". In: *Proceedings of the 27th International Conference on Software Engineering and Knowledge*

Engineering. SEKE 2015 (Wyndham Pittsburgh University Center, Pittsburgh, PA, USA, July 6–8, 2015). Ed. by Marek Reformat and Haiping Xu. Skokie, IL, USA: KSI Research Inc. and Knowledge Systems Institute Graduate School, 2015, pp. 260–265. ISBN: 1-891706-35-7. DOI: 10.18293/SEKE2015-9.

[143] Jiahui Wu, Zhengzi Xu, Wei Tang, Lyuye Zhang, Yueming Wu, Chengyue Liu, Kairan Sun, Lida Zhao, and Yang Liu. "OSSFP: Precise and Scalable C/C++ Third-Party Library Detection using Fingerprinting Functions". In: *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering*. ICSE 2023 (Melbourne, Australia, May 14–20, 2023). Ed. by John Grundy, Lori Pollock, and Massimiliano Di Penta. Los Alamitos, CA, USA: IEEE, 2023, pp. 270–282. ISBN: 978-1-6654-5701-9. DOI: 10.1109/ICSE48619.2023.00034.

[144] Weiwei Xu, Hao He, Kai Gao, and Minghui Zhou. "Understanding and Remediating Open-Source License Incompatibilities in the PyPI Ecosystem". In: *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2023 (Echternach, Luxembourg, Sept. 11–15, 2023). Ed. by Tegawendé F. Bissyandé, Jacques Klein, Christian Bird, and Federica Sarro. Los Alamitos, CA, USA: IEEE, 2023, pp. 178–190. ISBN: 979-8-3503-2996-4. DOI: 10.1109/ASE56229.2023.00175.

[145] Can Yang, Zhengzi Xu, Hongxu Chen, Xiaorui Gong, and Baoxu Liu. "Modx: Binary Level Partial Imported Third-Party Library Detection through Program Modularization and Semantic Matching". 2022. DOI: 10.48550/ARXIV.2204.08237. arXiv: 2204.08237 [cs.SE].

[146] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús M. González-Barahona. "An Empirical Analysis of Technical Lag in npm Package Dependencies". In: *New Opportunities for Software Reuse – 17th International Conference on Software Reuse, Proceedings*. ICSR 2018 (Madrid, Spain, May 21–23, 2018). Ed. by Rafael Capilla, Barbara Gallina, and Carlos Cetina. Lecture Notes in Computer Science 10826. Berlin, Germany: Springer, 2018, pp. 95–110. ISBN: 978-3-319-90420-7. DOI: 10.1007/978-3-319-90421-4_6.

[147] Ahmed Zerouali, Valerio Cosentino, Tom Mens, Gregorio Robles, and Jesús M. González-Barahona. "On the Impact of Outdated and Vulnerable Javascript Packages in Docker Images". In: *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*. SANER 2019 (Hangzhou, China, Feb. 24–27, 2019). Ed. by Xinyu Wang, David Lo, and Emad Shihab. Los Alamitos, CA, USA: IEEE, 2019, pp. 619–623. ISBN: 978-1-7281-0591-8. DOI: 10.1109/SANER.2019.8667984.

[148] Ahmed Zerouali, Tom Mens, and Coen De Roover. "On the usage of *JavaScript*, *Python* and *Ruby* packages in Docker Hub images". In: *Science of Computer Programming* 207, 102653 (June 2021): *Special issue on Software Health of Software Ecosystems*. Ed. by Dario Di Nucci, Eleni Constantinou, Raula Gaikovina Kula, and Henrique Rocha, 19 pp. ISSN: 1872-7964. DOI: 10.1016/J.SCICO.2021.102653.

[149] Ahmed Zerouali, Tom Mens, Alexandre Decan, and Coen De Roover. "On the impact of security vulnerabilities in the npm and RubyGems dependency networks". In: *Empirical Software Engineering* 27.5, 107 (Sept. 2022). Ed. by Jeffrey C. Carver, 45 pp. ISSN: 1573-7616. DOI: 10.1007/S10664-022-10154-1.

[150] Ahmed Zerouali, Tom Mens, Alexandre Decan, Jesús M. González-Barahona, and Gregorio Robles. "A multi-dimensional analysis of technical lag in Debian-based Docker images". In: *Empirical Software Engineering* 26.2, 19 (Mar. 2021): *Software Analysis, Evolution and Reengineering (SANER)*. Ed. by Emad Shihab and David Lo, 45 pp. ISSN: 1573-7616. DOI: 10.1007/S10664-020-09908-6.

[151] Ahmed Zerouali, Tom Mens, Gregorio Robles, and Jesús M. González-Barahona. "On the Relation between Outdated Docker Containers, Severity Vulnerabilities, and Bugs". In: *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*. SANER 2019 (Hangzhou, China, Feb. 24–27, 2019). Ed. by Xinyu Wang, David Lo, and Emad Shihab. Los Alamitos, CA, USA: IEEE, 2019, pp. 491–501. ISBN: 978-1-7281-0591-8. DOI: 10.1109/SANER.2019.8668013.

[152] Ahmed Zerouali, Ruben Opdebeeck, and Coen De Roover. "Helm Charts for Kubernetes Applications: Evolution, Outdatedness and Security Risks". In: *Proceedings of the 20th IEEE/ACM International Conference on Mining Software Repositories*. MSR 2023 (Melbourne, Australia, May 15–16, 2023). Ed. by Emad Shihab, Patanamon Thongtanunam, and Bogdan Vasilescu. Los Alamitos, CA, USA: IEEE, 2023, pp. 523–533. ISBN: 979-8-3503-1184-6. DOI: 10.1109/MSR59073.2023.00078.

[153] Ahmed Zerouali, Camilo Velázquez-Rodríguez, and Coen De Roover. "Identifying Versions of Libraries used in Stack Overflow Code Snippets". In: *Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories*. MSR 2021 (Virtual Event, Madrid, Spain, May 17–19, 2021). Ed. by Gregorio Robles, Kelly Blincoe, and Meiyappan Nagappan. Los Alamitos, CA, USA: IEEE, 2021, pp. 341–345. ISBN: 978-1-7281-8710-5. DOI: 10.1109/MSR52588.2021.00046.

[154] Yue Zhang, Fan Wu, and Akond Rahman. "Practitioner Perceptions of Ansible Test Smells". In: *Companion Proceedings of the 20th International Conference on Software Architecture*. 2nd International Workshop on the Foundations of Infrastructure Specification and Testing (FIST 2023) (L'Aquila, Italy, Mar. 14–17, 2023). Ed. by Luciano Baresi, Giovanni Quattrocchi, and Damian Andrew Tamburri. Los Alamitos, CA, USA: IEEE, 2023, pp. 325–327. ISBN: 978-1-6654-6459-8. DOI: 10.1109/ICSA-C57050.2023.00074.

[155] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Amit S. Warke, Mohamed Mohamed, and Ali Raza Butt. "Large-Scale Analysis of the Docker Hub Dataset". In: *Proceedings of the 2019 IEEE International Conference on Cluster Computing*. CLUSTER 2019 (Albuquerque, NM, USA, Sept. 23–26, 2019). Ed. by Patrick Bridges, Ron Brightwell, Pat McCormick, and Martin Schulz.

Los Alamitos, CA, USA: IEEE, 2019, pp. 1–10. ISBN: 978-1-7281-4734-5. DOI: 10.1109/CLUSTER.2019.8891000.