Control and Data Flow in Security Smell Detection for Infrastructure as Code: Is It Worth the Effort?

Ruben Opdebeeck
Ruben.Denzel.Opdebeeck@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Ahmed Zerouali
Ahmed.Zerouali@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Coen De Roover
Coen.De.Roover@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

***Abstract*—Infrastructure as Code is the practice of developing and maintaining computing infrastructure through executable source code. Unfortunately, IaC has also brought about new cyber attack vectors. Prior work has therefore proposed static analyses that detect security smells in Infrastructure as Code files. However, they have so far remained at a shallow level, disregarding the control and data flow of the scripts under analysis, and may lack awareness of specific syntactic constructs. These limitations inhibit the quality of their results. To address these limitations, in this paper, we present GASEL, a novel security smell detector for the Ansible IaC language. It uses graph queries on program dependence graphs to detect 7 security smells. Our evaluation on an oracle of 243 real-world security smells and comparison against two state-of-the-art security smell detectors shows that awareness of syntax, control flow, and data flow enables our approach to substantially improve both precision and recall. We further question whether the additional effort required to develop and run such an approach is justified in practice. To this end, we investigate the prevalence of indirection through control and data flow in security smells across more than 15 000 Ansible scripts. We find that over 55% of security smells contain data-flow indirection, and over 32% require a whole-project analysis to detect. These findings motivate the need for deeper static analysis tools to detect security vulnerabilities in IaC.**

***Index Terms*—Infrastructure as Code; Ansible; code smells; security; program dependence graph; empirical study**

I. Introduction

Infrastructure as Code (IaC) has emerged as the practice of automating the process of deploying and maintaining a digital infrastructure through executable source code [1]. A main benefit is that practitioners can apply software engineering principles for general-purpose code to infrastructure code. Examples include checking the code into version control systems and applying code quality assessments.

Part of the IaC practice is enabled by configuration management tools such as Ansible, Puppet, and Chef. Practitioners write scripts in these languages to automatically configure machines in their infrastructure, ranging from user account management, over installing software dependencies, to managing the software's configuration files. The aforementioned tools have gained considerable traction in recent years [2], with Ansible being the most popular today [3].

Nonetheless, since infrastructure code remains source code, it is plagued by the same problems as application code. Defects can cause issues in infrastructure deployments and potentially lead to outages [4]. Bad practices and code smells (i.e., recurring code patterns that indicate potential problems [5]) can hamper the maintainability of infrastructure code, decreasing the velocity at which changes are made [6]–[9]. Worse, security-related bad practices can give rise to security vulnerabilities in an infrastructure. Such vulnerabilities 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 written in Ansible, Chef, and Puppet [10]–[12]. Language-specific security smell detectors have been developed to this end. As the most popular IaC language, Ansible is supported by two tools, SLAC [11] and GLITCH [12], which detect various security smells such as hardcoded secrets and missing integrity checks for downloads.

However, we observe a number of limitations exhibited by existing approaches. Specifically, they lack awareness of Ansible syntax 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 being reported as safe (i.e., false negatives).

In this paper, we propose a novel approach to detecting security smells which leverages Program Dependence Graphs (PDG) built for Ansible code [9]. This representation includes control-flow and data-flow information, enabling our approach to address the latter two limitations. Moreover, we leverage Ansible's own parser when building the PDGs, thereby introducing more syntax awareness. We implement this approach in GASEL (Graph-based Ansible SEcurity Linter), a prototype detector for seven previously-proposed security smells.

We evaluate our approach on an oracle of 243 real-world security smells and show that it outperforms two state-of-the-art approaches. However, our approach is more expensive to both implement and apply. We therefore investigate whether the higher cost is justified in practice. To this end, we empirically investigate the prevalence of indirection caused by control flow and data flow in security smells found across a large corpus of Ansible scripts. Our findings show that these types of indirection are indeed common in practice, motivating the need for deeper static analysis tools for IaC languages.

To summarise, this paper makes the following contributions.

- We describe a PDG representation for both Ansible

playbooks (client) and role (library) code, based on Opdebeeck et al.'s prior work [9], which only supports the latter and a smaller subset of the Ansible syntax.

- We propose a novel PDG-based approach to security smell detection for IaC. In contrast to existing approaches, which consider individual files (akin to intra-procedural analysis), our approach performs a whole-project, inter-procedural analysis and leverages data flow to guide its smell detection.
- We construct an oracle of 243 real-world security smells, which is over 5 times larger than oracles from prior work.
- We evaluate our approach on this oracle, showing that it substantially outperforms state-of-the-art security smell detectors in both precision and recall.
- We empirically investigate the prevalence of control-flow and data-flow indirection within security smells in a dataset of over 15 000 Ansible scripts to motivate taking such information into account when developing static analysis tools for IaC.

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

II. Background

A. Ansible

Ansible scripts comprise a number of *tasks* describing the steps needed to configure an infrastructure machine. Each task executes an action, implemented as a module, which can be provided with arguments. Tasks also support various directives to enable conditional or looping execution. Tasks are assembled into *plays*, each of which targets a group of machines to be configured identically. Plays are further grouped into *playbooks*, which orchestrate deployments of different types of machines.

Users can define variables at various places in their code, e.g., for a single tasks, for all tasks in a play, etc. These variables can be referenced in expressions (denoted using double braces). Expression are often used to manipulate data.

Tasks and variables can further be modularised into reusable *roles*. These are akin to "packages" or "modules" in general-purpose languages. Roles can then be included into a play, which will define all its variables and execute all its tasks.

B. IaC Security Smells

The earliest work on security smells in infrastructure code is by Rahman et al. [10], 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. later replicate this work for Ansible and Chef [11], developing a similar tool named SLAC.

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,

```
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
```

Fig. 1. Task adapted from `personium/Ansible` exhibiting *HTTP Without SSL/TLS* and *Missing Integrity Check* smells.

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 et al. to create GLITCH [12], a polyglot detection tool for Puppet, Chef, and Ansible. Although the detection rules are largely identical to Rahman et al.'s, GLITCH only implements a single version operating on an intermediate representation. Saavedra et al. 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.

GLITCH supports the following security smells for Ansible: **Admin by default**: Specifying users with administrator privileges can violate the principle of least privilege. **Empty password**: Empty passwords are trivial to crack. **Hardcoded 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*. **HTTP without SSL/TLS**: 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**: When downloading executables, their integrity should be checked using cryptographic hashes. Unwanted modifications will otherwise go unnoticed. **Suspicious comment**: Comments such as "TODO" or "FIXME" may reveal weaknesses in the system. **Unrestricted IP address**: Having a server listen on `0.0.0.0` exposes the server to the whole network. Missing IP address filters facilitate denial of service attacks. **Weak crypto algorithm**: Hashing algorithms such as SHA-1 and MD5 are prone to collision attacks, e.g., on hashed passwords.

Finally, Reis et al. [13] 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.

III. Motivating Examples

The motivating examples depicted in Figures 1 and 2 highlight the 3 major limitations exhibited by SLAC and GLITCH.

```yaml
- hosts: ...
  roles:
    - role: overdrive3000.percona
      root_password: _password_

# In overdrive3000/percona role
- name: Update MySQL root password
  mysql_user:
    name: root
    password: '{{ root_password }}'
```

Fig. 2. Play adapted from `CenturyLinkCloud/clc-ansible-module` exhibiting an indirect *Hardcoded Secret* smell.

A. Lack of Ansible Syntax Awareness

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, exhibited in Figure 1. This notation denotes task arguments as a string where argument names and values are separated by equals signs, rather than as a nested dictionary (cf. Figure 2, lines 9–10). YAML parsing therefore does not suffice to create the key-value pairs required for existing tools to check these arguments for smells, leading them to miss the *HTTP Without SSL/TLS* and *Missing Integrity Check* smells exhibited in Figure 1. Moreover, since existing approaches do not fully support expressions, they may miss the smells even when standard YAML notation is used.

B. 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 Figure 2, for instance, a password is specified as an expression (line 10) that refers to a variable (line 4) initialised with a literal. 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.

C. Lack of Control-flow Information

Ansible projects are structured into separate YAML files which are dynamically included at run time. Moreover, they can dynamically include third-party code, as depicted in Figure 2, where a play (lines 1–4) dynamically includes a role from a third-party dependency. Existing approaches do not inspect the control flow 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 [11], 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 Figure 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 [9], detecting such indirect smells requires accurately tracking both control and data flow, and thus necessitates a whole-project analysis.

IV. Graph-based Security Smell Detection

To address the above limitations, we introduce an approach to detecting the previously-proposed security smells using graph queries on a Program Dependence Graph representation. The approach builds a PDG for an Ansible "entrypoint", either a playbook or a role (Section IV-A). We devise Cypher graph queries for each supported security smell which, when run on a PDG for an entrypoint, can detect the security smells that would occur when this entrypoint is executed (Section IV-B).

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.

A. Building Program Dependence Graphs

The PDGs produced by our approach are obtained using an extended version of the PDG builder proposed by Opdebeeck et al. [9]. Although the existing builder can accurately represent the control and data flow of Ansible code, it only supports Ansible roles. The lack of support for playbooks is a severe limitation, as playbooks are the code that clients actually execute. They are thus vital to consider when performing a whole-project analysis. Moreover, their PDG builder only supports simple tasks and a small subset of variables definitions. Among others, it lacks support for handlers (a special type of tasks), tasks which dynamically include roles, prompted variables, and any variable type related to playbooks. As these limitations cause it to miss large parts of Ansible codebases, we extend their PDG builder with support for these concepts.

Our PDG builder takes as input an Ansible playbook or role. It parses the input using Ansible's own parser, thereby supporting Ansible's extended syntax such as string-based task module arguments (cf. Section III-A). For playbooks, it traverses each play and creates a disconnected subgraph of the PDG to represent the play. Specifically, it explores each task and handler reachable from the play and constructs nodes to represent them. For roles, it analogously explores each task or handler that would be executed when the role is executed. Tasks are interconnected using control-flow edges (`order`) representing the order in which the tasks are executed.

While exploring the tasks, the builder keeps track of the variable definitions, which it uses to resolve variable references when it encounters expressions. It finds the latter by inspecting the task's directives and module arguments, through which it produces data nodes for expressions, variables, and literal values. It interlinks these data nodes through data-flow edges representing data definition (`def`) and usage (`use`), and connects the applicable data nodes to the task via usage edges labelled with the name of the directive or module argument. Throughout this process, the builder accounts for the many intricacies of Ansible's variable precedence and expression evaluation. For a more detailed description, we refer the interested reader to Opdebeeck et al. [9].

Some special cases are handled differently. Tasks that lead to the dynamic inclusion of Ansible files are handled by resolving the file that would be included, if possible, and expanding the PDG with new nodes created for the included file. For

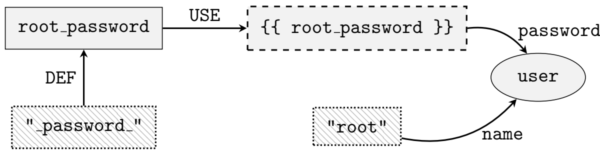

Fig. 3. PDG of the example depicted in Figure 2.

variable includes (e.g., `include_vars`), the variables defined in this file are tracked analogously to other variables. For task includes (e.g., `include_tasks`), the builder adds each included task to the PDG analogously to other tasks. For role includes (e.g., `include_role`), the builder applies the same building process it does for input roles, yet also uses the previously-tracked variables since the role may use variables defined outside of its scope. The builder searches for the included role within the repository itself (for first-party roles) and in a configurable search directory (for third-party roles). Throughout the handling of these special cases, the builder uses control-flow information to ensure the PDG only contains nodes for tasks that are actually reachable from a playbook or role, and thus omits irrelevant files. This enables our approach to overcome the limitation described in Section III-C.

Figure 3 depicts the PDG built for the example of Figure 2. The sole task is depicted as an oval on the right. It is linked to its two module arguments via usage edges, leading to a definition-use chain of an expression (dashed rectangle), variable (solid rectangle), and a literal (dotted rectangle). This chain from the literal to the password argument is indicative of an indirectly hardcoded secret.

B. Security Smell Detection

To detect security smells, our approach leverages graph queries written in the Cypher query language, which are run on the PDG constructed by the builder. We design a Cypher query for 7 of the 8 smells supported by GLITCH (see Section II-B). We do omit the *Suspicious comment* smell, as comments are missing from 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 Figure 2, thus addressing the limitation described in Section III-B.

Table I summarises our graph queries for security smell detection. Figure 4 depicts an example of a corresponding Cypher query. In the table, the function `hasDUPath`(n_1, e, n_2) finds definition-use paths between nodes n_1 and n_2, where the final edge in the path is denoted by e. A second form with two arguments, `hasDUPath`(n_1, n_2) is used when the final edge is irrelevant. Functions `isExpression` and `isLiteral` check the type of a given node, while functions such as `isPassword` or `isUser` check the label of a given node or edge. Similarly

```
1 MATCH (l:Literal)-[:DEF|USE*0..]->()-[e:KEYWORD]->(t:Task)
2 WHERE (e.value CONTAINS "user" OR e.value CONTAINS "role")
3     AND (l.value = "admin" OR l.value = "root")
4 RETURN l.location;
```

Fig. 4. Simplified graph query for *Admin By Default* smell.

to SLAC and GLITCH, the latter functions use string patterns, summarised in Table II.

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 when 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. [13].

V. Security Smells in Practice

The following research questions evaluate our approach and assess the need for deep static analysis in smell detection:

- RQ_1: *How accurate is our security smell detector?*
- RQ_2: *How prevalent are security smells in open-source Ansible codebases?*
- RQ_3: *How often do security smells cross file boundaries?*
- RQ_4: *How prevalent is data-flow indirection in security smells?*

A. Dataset Collection

To answer these questions, we need a dataset of Ansible repositories. We start from the dataset previously collected by Saavedra et al. [12]. They collected a list of 681 repositories through a GitHub search and applied various filtering criteria focused on development characteristics (number of contributors, number of commits) to discard irrelevant projects. However, we argue that these criteria may have discarded several relevant, frequently-used repositories, while simultaneously retaining irrelevant projects. We aim to enrich this initial dataset by adding relevant repositories that were missed, while also removing irrelevant repositories that are rarely used.

First, we use the Ansible Galaxy registry[1] to add popular open-source reusable Ansible roles to the dataset. To this end, we use the Andromeda dataset [14] to collect the most popular repositories that cumulatively represent 95% of all role downloads in the 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.

Then, we remove repositories with no GitHub stars, which indicates they lack popularity and are thus irrelevant for our purposes. This removes 272 repositories from the dataset, 199 of which originate from the original dataset by Saavedra et al. Moreover, we remove 49 hidden forks, 34 of which originate from the original dataset. These are repositories which are not

[1]https://galaxy.ansible.com/

TABLE I
DETECTION RULES FOR SECURITY SMELLS. n DENOTES ANY NODE, l, v, AND t DENOTE LITERAL, VARIABLE, AND TASK NODES, e DENOTES EDGES.

Smell type	Query description
Admin by default	$\texttt{hasDUPath}(l,e,t) \land \texttt{isAdmin}(l) \land \texttt{isUser}(e)$
Empty password	$[(\texttt{hasDUPath}(l,e,t) \land \texttt{isPassword}(e)) \lor (\texttt{hasDUPath}(l,v) \land \texttt{hasDUPath}(v,t) \land \texttt{isPassword}(v))] \land \texttt{isEmpty}(l)$
Hardcoded secret	$[(\texttt{hasDUPath}(l,e,t) \land \texttt{isSecret}(e) \land \neg\texttt{isSecretWhitelist}(e)) \lor (\texttt{hasDUPath}(l,v) \land \texttt{hasDUPath}(v,t) \land \texttt{isSecret}(v) \land \neg\texttt{isSecretWhitelist}(v))] \land \neg\texttt{isEmpty}(l)$
HTTP without SSL/TLS	$\texttt{hasDUPath}(n,t) \land (\texttt{isLiteral}(n) \lor \texttt{isExpression}(n)) \land \texttt{isHTTP}(n) \land \neg(\texttt{isHTTPWhitelist}(n) \lor (\texttt{hasDUPath}(l,n) \land \texttt{isHTTPWhitelist}(l)))$
Missing integrity check	$[\texttt{hasDUPath}(l_1,t) \land \texttt{isDownload}(l_1) \land \neg(\texttt{hasDUPath}(n,e_1,t) \land \texttt{isChecksum}(e))] \lor [\texttt{hasDUPath}(l_2,e_2,t) \land ((\texttt{isCheckFlag}(e_2) \land \neg l_2) \lor (\texttt{isDisableCheckFlag}(e_2) \land l_2))]$
Unrestricted IP address	$\texttt{hasDUPath}(n,t) \land (\texttt{isLiteral}(n) \lor \texttt{isExpression}(n)) \land \texttt{isBadIP}(n)$
Weak crypto algorithm	$\texttt{hasDUPath}(n,t) \land (\texttt{isLiteral}(n) \lor \texttt{isExpression}(n)) \land \texttt{isWeakCrypto}(n)$

TABLE II
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
`isDisableCheckFlag`	disable_gpg_check
`isBadIP`	0.0.0.0
`isWeakCrypto`	md5, sha1, arcfour

TABLE III
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

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 entrypoints (playbooks or roles). A summary of this dataset is depicted in Table III.

B. Empirical Analysis

We describe the research method and the results for each research question separately.

RQ_1: How accurate is our security smell detector?: This RQ serves as the evaluation of our approach, in which 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 [11] and GLITCH [12].

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, it does not serve our needs for two 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.

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 [15] 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 Andromeda dataset [14]. 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 IV-B.

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 less 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.

The first author, who has 3 years of experience with Ansible and security, then manually labels 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.

Table IV 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 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 $\mathbf{RQ_1}$**: 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*.

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, entrypoints, and files. The results will provide insights into the prevalence of security smell types in practice.

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 entrypoint separately, it may report duplicates when smells are reachable from multiple entrypoints in a repository, e.g., because both entrypoints include a common file. Since duplicate reports would skew our results, we eliminate them by not considering the entrypoint through which the smells are reported, except when investigating the proportion of entrypoints affected by smells.

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

These smells are spread across 3 145 source files, affecting 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, entrypoint, and sink file are summarised using boxen plots in Figure 5. 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 boxen plot. Indeed, we found repositories with as many as 802 smells. Similarly, the maximum number of smells in a single entrypoint is 247.

Table V 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

TABLE IV
PRECISION AND RECALL FOR GASEL, SLAC, AND GLITCH ON THE ORACLE DATASET.

Smell type	# instances	GASEL Precision	GASEL Recall	GLITCH Precision	GLITCH Recall	SLAC Precision	SLAC Recall
Admin By Default	64	98.11%	**81.25%**	**100.00%**	67.19%	*N/A*	*N/A*
Empty Password	15	**44.44%**	**80.00%**	42.86%	60.00%	30.77%	53.33%
HTTP Without SSL/TLS	35	**100.00%**	**88.57%**	54.84%	48.57%	22.89%	54.29%
Hardcoded Secret	11	45.45%	**90.91%**	**56.25%**	81.82%	33.33%	81.82%
Missing Integrity Check	27	**96.15%**	**92.59%**	50.00%	33.33%	75.00%	44.44%
Unrestricted IP Address	47	76.60%	76.60%	**82.98%**	82.98%	81.63%	**85.11%**
Weak Crypto Algorithm	44	**97.67%**	**95.45%**	86.11%	70.45%	*N/A*	*N/A*

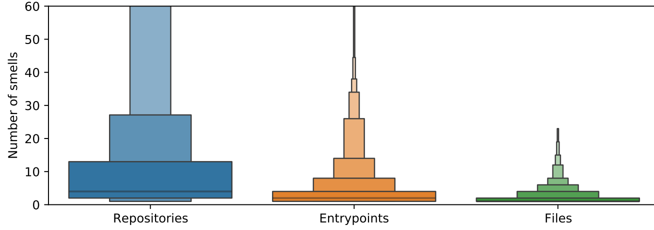

Fig. 5. Distribution of the number of smells per repository, entrypoint and sink file.

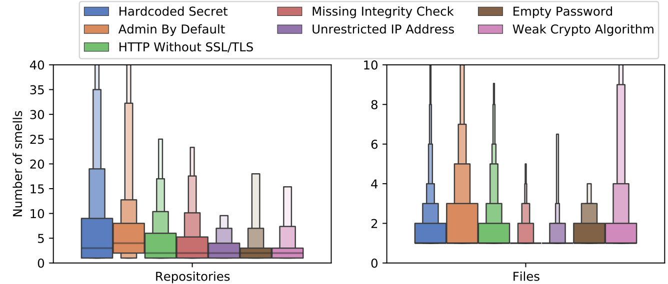

Fig. 6. Distribution of the number of smells per repository and sink file, grouped by smell type.

TABLE V
NUMBER OF SMELLS, AFFECTED REPOSITORIES AND SINK FILES GROUPED BY SMELL TYPE.

Smell type	# smells	% smells	# files	# repos
Hardcoded Secret	2 155	27.17	1 235	211
Admin By Default	2 043	25.75	766	190
HTTP Without SSL/TLS	1 369	17.26	765	198
Missing Integrity Check	1 084	13.66	803	200
Weak Crypto Algorithm	577	7.27	234	94
Unrestricted IP Address	418	5.27	289	120
Empty Password	287	3.61	163	65

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 repositories or files often contain multiple *Admin By Default* smells while *Missing Integrity Check* more often occurs alone. The table also suggests that smell types are not evenly distributed across all repositories, since the highest number of repositories affected by a smell type (211) is less than half of all affected repositories (472).

Finally, Figure 6 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 are affected by the smell type, so the minimum number in each is 1. We observe that the median number of smells per entrypoint 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 often have multiple *Admin By Default* instances, explaining the observation described above.

Answer to $\mathbf{RQ_2}$: 49% and 22% of the repositories and entrypoints 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*.

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 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.

Research Method: 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 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 $\mathbf{RQ_3}$: 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.

RQ_4: How prevalent is data-flow indirection in security smells?: Our final RQ investigates how often 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.

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.

Results: We find that 55.5% (4 402) of the detected smells involve some level of indirection through variables and expressions. Table VI 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.

Figure 7 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 VI
NUMBER OF SMELLS, AFFECTED REPOSITORIES AND SINK FILES GROUPED BY SMELL DATA-FLOW INDIRECTION LEVEL.

Indirection	# smells	% smells	# files	# repos
0	3 531	44.51	1 759	326
1	3 227	40.68	1 565	304
2	957	12.06	456	134
3	181	2.28	70	36
4	17	0.21	11	10
5	13	0,16	7	7
6	7	0,09	3	2

Fig. 7. A heatmap showing the proportion of smells of each group (by type) and its indirection level.

Answer to $\mathbf{RQ_4}$: 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.

VI. 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.

A. 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.

1) 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 III-A) contributes substantially to the improved recall for *Admin By Default* and *HTTP Without SSL/TLS* smells. Among others, the smell exemplified in Figure 1 was found by GASEL but missed by other tools. Furthermore, awareness of the `register` directive, whose value is a variable name used to store the outcome of a task,

```
- 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
```

Fig. 8. Simplified example of a false *HTTP Without SSL/TLS* smell avoided using data flow, adapted from `sodafoundation/installer`.

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

Fig. 9. Simplified example of a false *Hardcoded Secret* smell with data-flow indirection, adapted from `kubeinit/kubeinit`.

enabled our approach to avoid numerous false positives of *Weak Crypto Algorithm*, for which GLITCH reported instances because of the use of `md5` in the variable name[2].

2) Data-flow Information: Support for expressions and indirection uncovered 8 new instances of *Missing Integrity Check* smells. Moreover, GASEL uncovered one new instance each for *Admin By Default*, *Hardcoded Secret*, and *Weak Crypto Algorithm*, among which the motivational example depicted in Figure 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 Figure 8. 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 Figure 9. Although the content of such files is secret, the reported values only contain the path and are thus false positives. Further improvements to the string patterns may aid in avoiding these false positives.

3) 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 because there exists control flow which prevents the empty password from being used, either by skipping tasks or asserting that a variable is not empty. 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 *Unrestricted IP Address*. Here, the `0.0.0.0` IP address is used in a deny-rule in a firewall configuration, and is thus the opposite of the associated security weakness. Future work should investigate taking more contextual information into account to avoid such false positives.

4) 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. 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. Further refining the string patterns 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 ML 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.

5) 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 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 of 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 further.

B. 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.

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

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 a large number of false positive reports. A majority of such files were test files, which developers often consider irrelevant [11]. Moreover, we observe that SLAC and GLITCH scanned Ansible files that are never included through an entrypoint, 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.

C. On the Importance of Control and Data Flow

Our investigation suggests that over half of 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 (Section VI-A) shows that taking data-flow information into account can lead to substantial improvements in both precision and recall, and we therefore suggest future research to follow this direction.

Although indirection through control flow is less prevalent (see RQ_4), 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 (Section VI-B). Finally, control-flow information would be vital to address the low precision for *Empty Password* smells (Section VI-A3).

We note that *Missing Integrity Check*, which was the least prevalent smell according to Saavedra et al. [12], 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 [13]. 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.

D. Threats to Validity

We present our threats to validity according to the recommendations of Wohlin et al. [16]. 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 VI-B). 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.

VII. Related Work

Research on IaC has steadily increased over the years. An early systematic mapping study [17] categorises research into four topics, namely tooling [18], adoption of IaC, empirical studies [2], [19], [20], and testing [21], [22]. More recently, Chiari et al. survey the literature for static analysis approaches for IaC [23], and found that the majority of approaches focus on code smell and anti-pattern detection [6]–[8], [24], defect prediction [25], [26], or model checking [27], [28].

For Ansible specifically, various approaches to analysing its code have been proposed. Dalla Palma et al. perform defect prediction using a large catalogue of software quality metrics for Ansible [26]. Borovits et al. use deep learning to detect anti-patterns and inconsistencies in the naming of Ansible tasks [29]. Opdebeeck et al. apply change distilling of Ansible code to study and predict version increments in open-source Ansible roles [30]. Kokuryo et al. investigate the use of imperative modules in Ansible roles [31]. Later, Horton and Parnin propose a dynamic analysis approach to automatically migrate imperative modules to declarative ones [32]. Dai et al. extract shell commands from Ansible code using a

structured representation, which they then check using a shell security linter to detect risky scripts [33]. The aforementioned SLAC tool by Rahman et al. checks for security smells in Ansible [11], while Saavedra et al.'s GLITCH tool proposed an intermediate representation to perform the same task [12]. Finally, Opdebeeck et al. propose a Program Dependence Graph representation to check suspicious usages of variables in Ansible roles [9].

Various researchers have focused on security in IaC. The aforementioned works by Rahman et al. [10], [11] and Saavedra et al. [12] investigate security smells in configuration management languages. Reis et al. use feedback from practitioners to improve Rahman et al.'s security linter for Puppet [13]. As described above, Dai et al. detect risky shell commands in Ansible [33]. Lepiller et al. model the data flow and security assumptions of CloudFormation templates to detect so-called intra-update sniping vulnerabilities [34]. Kumara et al. [35] model TOSCA templates as knowledge graphs and use SPARQL graph queries to detect, among others, the security smells proposed by Rahman et al. Although the two latter works are similar to ours, we detect security smells in Ansible, which is a more general automation language. Moreover, we use data-flow information to guide smell detection and account for data indirection, which, to the best of our knowledge, has not yet been considered in IaC research.

VIII. Conclusion

Prior approaches to detecting security smells in Infrastructure as Code disregard the analysed scripts' control and data flow, and lack awareness of specific syntactic constructs. To address these limitations, we presented an approach based on Program Dependence Graphs to detect 7 security smells in Ansible code. The PDG provides vital data-flow information and can account for Ansible's syntactic particularities. Moreover, its construction follows the control flow of Ansible scripts, enabling our approach to 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 entrypoints. We found that over half of security smells involve data-flow indirection, and one in three smells involve dynamic file inclusion. These findings strengthen the motivation to include control-flow and data-flow information into future security smell detection approaches.

Acknowledgements

This research was partially funded by the "Cybersecurity Initiative Flanders" project and the Research Foundation Flanders (FWO) under Grant No. 1SD4321N. We thank Arne Van Quickelberghe for his efforts to conduct an initial investigation that made this work possible.

References

[1] K. Morris, *Infrastructure as Code: Managing Servers in the Cloud*, 1st ed. O'Reilly, 2016.

[2] M. Guerriero, M. Garriga, D. A. Tamburri, and F. 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, Industrial Track*, ser. ICSME '19, 2019, pp. 580–589.

[3] StackExchange, Inc. (2022) 2022 annual stackoverflow developer survey. [Online]. Available: https://survey.stackoverflow.co/2022/#section-most-popular-technologies-other-tools

[4] A. Rahman, E. Farhana, C. Parnin, and L. Williams, "Gang of eight: A defect taxonomy for infrastructure as code scripts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, 2020, pp. 752–764.

[5] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[6] J. Schwarz, A. Steffens, and H. Lichter, "Code smells in infrastructure as code," in *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, 2018, pp. 220–228.

[7] E. van der Bent, J. Hage, J. Visser, and G. 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*, ser. SANER '18, 2018, pp. 164–174.

[8] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16, 2016, pp. 189–200.

[9] R. Opdebeeck, A. Zerouali, and C. De Roover, "Smelly variables in ansible infrastructure code: Detection, prevalence, and lifetime," in *Proceedings of the 19th International Conference on Mining Software Repositories*, ser. MSR '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 61–72.

[10] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19, 2019, pp. 164–175.

[11] A. Rahman, M. R. Rahman, C. Parnin, and L. Williams, "Security smells in Ansible and Chef scripts: A replication study," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 1, Jan. 2021.

[12] N. Saavedra and J. a. F. Ferreira, "GLITCH: Automated polyglot security smell detection in infrastructure as code," in *37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE22. New York, NY, USA: Association for Computing Machinery, 2022.

[13] S. Reis, R. Abreu, M. d'Amorim, and D. Fortunato, "Leveraging practitioners' feedback to improve a security linter," in *37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE22. New York, NY, USA: Association for Computing Machinery, 2022.

[14] R. Opdebeeck, A. Zerouali, and C. 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*, ser. MSR '21, 2021, pp. 580–584.

[15] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[16] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering - An Introduction*. Kluwer, 2000.

[17] A. Rahman, R. Mahdavi-Hezaveh, and L. A. Williams, "A systematic mapping study of infrastructure as code research," *Inf. Softw. Technol.*, vol. 108, pp. 65–77, Apr. 2019.

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

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

[20] I. Kumara, M. Garriga, A. U. Romeu, D. Di Nucci, F. Palomba, D. A. Tamburri, and W.-J. van den Heuvel, "The do's and don'ts of infrastructure code: A systematic gray literature review," *Inf. Softw. Technol.*, vol. 137, Sep. 2021.

[21] W. Hummer, F. Rosenberg, F. Oliveira, and T. Eilam, "Testing idempotence for infrastructure as code," in *Proceedings of the 14th ACM/IFIP/USENIX International Middleware Conference*, ser. Middleware '13, 2013, pp. 368–388.

[22] K. Ikeshita, F. Ishikawa, and S. Honiden, "Test suite reduction in idempotence testing of infrastructure as code," in *Proceedings of the 11th International Conference on Tests and Proofs*, ser. TAP@STAF '17, 2017, pp. 98–115.

[23] M. Chiari, M. De Pascalis, and M. Pradella, "Static analysis of infrastructure as code: a survey," in *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, 2022, pp. 218–225.

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

[25] A. Rahman and L. A. Williams, "Source code properties of defective infrastructure as code scripts," *Inf. Softw. Technol.*, vol. 112, pp. 148–163, Aug. 2019.

[26] S. Dalla Palma, D. Di Nucci, F. Palomba, and D. A. Tamburri, "Toward a catalog of software quality metrics for infrastructure code," *J. Syst. Softw.*, vol. 170, Dec. 2020.

[27] R. Shambaugh, A. Weiss, and A. Guha, "Rehearsal: A configuration verification tool for Puppet," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16, 2016, pp. 416–430.

[28] T. Sotiropoulos, D. Mitropoulos, and D. Spinellis, "Practical fault detection in Puppet programs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, 2020, pp. 26–37.

[29] N. Borovits, I. Kumara, D. Di Nucci, P. Krishnan, S. Dalla Palma, F. Palomba, D. A. Tamburri, and W.-J. v. d. Heuvel, "FindICI: Using machine learning to detect linguistic inconsistencies between code and natural language descriptions in infrastructure-as-code," *Empirical Software Engineering*, vol. 27, no. 7, pp. 1–30, 2022.

[30] R. Opdebeeck, A. Zerouali, C. Velázquez-Rodríguez, and C. De Roover, "On the practice of semantic versioning for Ansible Galaxy roles: An empirical study and a change classification model," *J. Syst. Softw.*, vol. 182, Dec. 2021.

[31] S. Kokuryo, M. Kondo, and O. Mizuno, "An empirical study of utilization of imperative modules in Ansible," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, 2020, pp. 442–449.

[32] E. Horton and C. Parnin, "Dozer: Migrating shell commands to Ansible modules via execution profiling and synthesis," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2022, pp. 147–148.

[33] T. Dai, A. Karve, G. Koper, and S. Zeng, "Automatically detecting risky scripts in infrastructure code," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20, 2020, pp. 358–371.

[34] J. Lepiller, R. Piskac, M. Schäf, and M. Santolucito, "Analyzing infrastructure as code to prevent intra-update sniping vulnerabilities," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2021, pp. 105–123.

[35] I. Kumara, Z. Vasileiou, G. Meditskos, D. A. Tamburri, W.-J. Van Den Heuvel, A. Karakostas, S. Vrochidis, and I. Kompatsiaris, "Towards semantic detection of smells in cloud infrastructure code," in *Proceedings of the 10th International Conference on Web Intelligence, Mining and Semantics*, ser. WIMS 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 63–67.