Noname manuscript No.
(will be inserted by the editor)

An Empirical Study of Task Infections in Ansible Scripts

Akond Rahman · Dibyendu Brinto Bose · Yue Zhang · Rahul Pandita

Received: date / Accepted: date

Abstract *Context*: Despite being beneficial for managing computing infrastructure at scale, Ansible scripts include security weaknesses, such as hard-coded passwords. Security weaknesses can propagate into tasks, i.e., code constructs used for managing computing infrastructure with Ansible. Propagation of security weaknesses into tasks makes the provisioned infrastructure susceptible to security attacks. A systematic characterization of task infection, i.e., the propagation of security weaknesses into tasks, can aid practitioners and researchers in understanding how security weaknesses propagate into tasks and derive insights for practitioners to develop Ansible scripts securely.

Objective: *The goal of the paper is to help practitioners and researchers understand how Ansible-managed computing infrastructure is impacted by security weaknesses by conducting an empirical study of task infections in Ansible scripts.*

Methodology: We conduct an empirical study where we quantify the frequency of task infections in Ansible scripts. Upon detection of task infections, we apply qualitative analysis to determine task infection categories. We also conduct a survey with 23 practitioners to determine the prevalence and sever-

Akond Rahman
Auburn University
E-mail: akond.rahman.buet@gmail.com

Dibyendu Brinto Bose
Virginia Tech
E-mail: brintodibyendu@vt.edu

Yue Zhang
Auburn University
E-mail: yzz0229@auburn.edu

Rahul Pandita
GitHub
E-mail: rahulpandita@github.com

ity of identified task infection categories. With logistic regression analysis, we identify development factors that correlate with presence of task infections.

Results: In all, we identify 1,805 task infections in 27,213 scripts. We identify six task infection categories: anti-virus, continuous integration, data storage, message broker, networking, and virtualization. From our survey, we observe tasks used to manage data storage infrastructure perceived to have the most severe consequences. We also find three development factors, namely age, minor contributors, and scatteredness to correlate with the presence of task infections.

Conclusion: Our empirical study shows computing infrastructure managed by Ansible scripts to be impacted by security weaknesses. We conclude the paper by discussing the implications of our findings for practitioners and researchers.

Keywords ansible · configuration script · devops · devsecops · empirical study · infrastructure as code · security

1 Introduction

Cloud service vendors, such as Amazon, provide multiple services to set up and manage computing infrastructure, such as EC2 instances and Amazon Elastic Container Service (ECS). These services are leveraged by organizations to deploy software applications at scale. While using these services practitioners need to provide necessary configurations and instructions on how to setup and manage these computing infrastructure.

With the practice of infrastructure as code (IaC), practitioners can create and manage computing infrastructure at scale (Humble and Farley, 2010). Ansible is a popular tool to implement the practice of IaC (Rahman et al., 2021b; Dalla Palma et al., 2022). Use of Ansible scripts have resulted in a lot of benefits for organizations (RedHat, 2022b,a; Rahman et al., 2020b; Mohammad Mehedi and Rahman, 2022). For example, use of Ansible scripts was one of the contributing factors for NetApp to reduce the software delivery time from days to seconds (RedHat, 2022b). As another example, use of Ansible scripts was one of the contributing factors for NEC to experience an increase in revenue for its product called 'NEC SDN Solution', as it helped in automating time consuming and labor-prone manual tasks (RedHat, 2022a).

On the one hand, Ansible scripts yield benefits for organizations in managing computing infrastructure, but on the other hand, unmitigated security weaknesses in Ansible scripts can be leveraged by malicious users to cause serious consequences at scale. Researchers (Rahman et al., 2021b; Rahman and Williams, 2021) have found Ansible scripts to include security weaknesses, such as hard-coded passwords. For example, Rahman et al. (2021b) identified 1,074 hard-coded passwords in 14,253 open-source Ansible scripts.

The prevalence of such security weaknesses in Ansible scripts necessitates an understanding of how security weaknesses affect Ansible-based infrastructure management. Organizations often use Ansible scripts to set up the com-

puting infrastructure necessary to run their businesses. Existing security weaknesses in Ansible scripts can propagate into computing infrastructure, leaving the provisioned computing infrastructure susceptible to security attacks. While secure development of Ansible scripts has garnered interest amongst researchers (Rahman et al., 2021b; Rahman and Williams, 2021; Hortlund, 2021; Rahman et al., 2021a), understanding how security weaknesses propagate into tasks and impact Ansible-based infrastructure management remains an under-explored area.

One approach to investigating the impact of security weaknesses on Ansible-based infrastructure management is to understand how security weaknesses propagate into tasks. A 'task' is a code construct that performs infrastructure management operations (Opdebeeck et al., 2022; Ansible, 2020; Borovits et al., 2022). With tasks, Ansible users can specify what configurations are required to set up and manage necessary computing infrastructure and how such configurations will be executed (Opdebeeck et al., 2022; Ansible, 2020; Borovits et al., 2022). If a security weakness propagates into a task, then we can determine that task to be impacted by that security weakness. We define the phenomenon of security weaknesses propagating into a task as 'task infection.' The code snippet in Figure 1, which is obtained from an open source software (OSS) repository (redhat performance, 2022), shows an example of task infection. We observe two security weaknesses: one hard-coded username and one hard-coded password. The hard-coded user name `sat_user` and the hard-coded password `sat_pass` propagate into a task called "`Determine organization ID`". In the figure, 'satellite6' is a hard-coded string but not a hard-coded secret, as this hard-coded string is not used to set up usernames, user passwords, or private tokens used for user authorization. We hypothesize that we can detect and characterize such task infections in Ansible scripts through systematic analysis. Such analysis can aid practitioners and researchers in understanding the nature of task infections. Further, from our empirical study, we can derive insights on how to develop Ansible scripts securely.

The goal of the paper is to help practitioners and researchers understand how Ansible-managed computing infrastructure is impacted by security weaknesses by conducting an empirical study of task infections in Ansible scripts.

Accordingly, we answer the following research questions:

- ***RQ1*: How frequently do task infections occur in Ansible scripts?**
  Motivation: The motivation for answering this research question is to empirically determine to what extent task infections exist in Ansible scripts. This is the first step toward characterizing task infections.
  Insight: We identify 1,805 of 49,898 tasks to experience task infection in our dataset of 27,213 Ansible scripts.
- ***RQ2*: What categories of task infections appear for Ansible scripts?**

  Motivation: The motivation for answering this research question is to develop a taxonomy for task infections. Such categorization can help the Ansible community understand what are the typical infrastructure-related

```
1 sat_user: admin
2 sat_pass: admin
3 ...
4 tasks:
5 - name: "Determine organization ID"
6   uri:
7     url: https://{{ groups['satellite6']|first
        ↪ }}/katello/api/organizations
8     method: GET
9     user: "{{ sat_user }}"
10    password: "{{ sat_pass }}"
11    force_basic_auth: yes
```

Hard-coded user name

Hard-coded password

Fig. 1: An example to demonstrate task infection. Two hard-coded secrets namely, `sat_user` and `sat_pass` propagate into the task `Determine organization ID`, causing a task infection.

tasks impacted by security weaknesses.
Insight: We identify six categories of task infections: antivirus, continuous integration, data storage, message broker, networking, and virtualization.

– ***RQ3*: What are the practitioner perceptions of task infection categories for Ansible scripts?**
Motivation: The motivation of answering this research question is to understand the practitioner's perceptions regarding the frequency and severity of task infection. Such understanding can grow the science of secure Ansible script development and also lay the groundwork for future research studies.
Insight: We observe 50% of the survey participants to find data storage-related tasks to be impacted severely.

– ***RQ4*: What development factors correlate with task infections in Ansible scripts?**
Motivation: The motivation for answering this research question is to help practitioners in prioritizing inspection efforts by quantifying the correlation between development activity metrics and task infections.
Insight: We identify 3 development activity metrics namely, age, minor contributors, and scatteredness to show correlation with task infection in Ansible scripts.

– ***RQ5*: How accurate is TIDAL in automatically identifying task infections in Ansible scripts?**
Motivation: The motivation for answering this research question is to develop a technique to automatically detect task infections. Such technique may aid practitioners in understanding the relevance of detected security

weaknesses, as practitioners query about how security weaknesses that are detected by static analysis tools are used within the code (Smith et al., 2015)
Insight: We construct a tool called `TIDAL` that uses data flow analysis to determine if a security weakness is used by a task. `TIDAL`'s precision and recall to detect task infections is > 0.85 across all studied security weakness categories.

We conducted an empirical study with 27,213 Ansible scripts mined from OSS repositories. We construct Task Infection Detector for Ansible Scripts (`TIDAL`) that we use to detect task infections and compute the frequency of infected tasks. Next, with open coding (Saldaña, 2015), we categorize tasks infected by security weaknesses. We also surveyed 23 practitioners to assess their perceptions of the identified task infection categories. Finally, using logistic regression (Long and Freese, 2006) we quantify the correlation between development factors and the presence of task infections in Ansible scripts. The dataset and source code used in our empirical study are available online (Rahman, 2023).

Contributions: We list our contributions as follows:

- A categorization of task infections in Ansible scripts;
- An analysis of how frequently task infections appear in Ansible scripts;
- An evaluation of practitioner perceptions for identified categories of task infections; and
- An empirical evaluation of what development factors correlate with task infection.

Motivation of this research study in the context of existing related work : Security static analysis of IaC scripts has gained interest amongst researchers, which in turn have generated multiple publications (Rahman et al., 2021b; Rahman and Parnin, 2023). We build on top of existing research conducted by Rahman et al. (2021b) which provided a taxonomy of security weaknesses in Ansible scripts. They (Rahman et al., 2021b) further quantified the frequency of detected security weaknesses with feedback from practitioners. We take motivation from their (Rahman et al., 2021b) research and characterize the propagation of security weaknesses in Ansible tasks, which we refer to as 'task infections'. Such characterization is important for the following reasons:

- *first*, in Ansible tasks are pivotal to manage computing infrastructure, and therefore with task infection characterization, we can understand if security weaknesses propagate into Ansible-based infrastructure management with tasks;
- *second*, in the case of static analysis, relevance, i.e., how detected alerts are used in code is important to practitioners (Smith et al., 2015). While Rahman et al. (2021b)'s paper provides a tool called SLAC to identify security weaknesses, it does not report which tasks exactly use the detected

security weaknesses. Hence, a systematic investigation is required to report to the practitioners what tasks are being impacted by security weaknesses, so that practitioners are motivated to take action on resolving reported weaknesses; and
- *third*, development activity metrics have not been studied in the context of secure Ansible script development. Existing research has not studied the relationships between development activity metrics and task infections. Our empirical study addresses this gap in IaC-related research.
- *fourth*, development of a new tool that will perform variable-aware syntax analysis and data flow analysis in order to detect task infections. 'SLAC', the tool developed by Rahman et al. (2021b), does not perform variable-aware syntax analysis and data flow analysis.

To date, the paper authored by Rahman and Parnin (2023) is the first to report security weaknesses propagation in the domain of IaC. In particular, they studied how security weakness propagates into `resources`, which are used to manage computing infrastructure with Puppet. One approach could have been to directly use their provided taxonomy but there are differences between Puppet and Ansible that necessitate systematic investigation to study security weakness propagation in the context of Ansible scripts. These differences are:

- *Agent requirements*: In the case of Puppet, for each server an additional agent needs to be installed. Ansible does not require the installation of such agents.
- *Execution order*: For Puppet, the current code state provides a clear view of what will be the configurations of the provisioned infrastructure. In the case of Ansible, the execution order is important as specifying a different order might provision the desired infrastructure incorrectly.
- *Perceived maintenance*: Practitioners (Yevgeniy Brikman, 2016) perceive Ansible-based scripts to incur more maintenance overhead. The state of the provisioned infrastructure might change constantly, and code written a week ago might become unusable, and practitioners have to write more code. In the case of Puppet, code represents the current state, and there may not be a need to write new scripts to be consistent with the current state.
- *State reconciliation*: In the case of Puppet state reconciliation is performed with `resources`, whereas for Ansible it is `task`.
- *Syntax*: Ansible scripts follow a procedural style, whereas Puppet scripts follow a declarative style.

All of these above-mentioned differences require a systematic investigation of how security weaknesses propagate into tasks for Ansible scripts. To conduct this empirical investigation, we adopt a differentiated replication research study (Krein and Knutson, 2010) of Rahman and Parnin (Rahman and Parnin, 2023). A differentiated replication study is where researchers change one or multiple aspects of the original study to answer similar research questions (Krein and Knutson, 2010). In any empirical science, replication is considered as a pivotal activity to develop new knowledge (Da Silva et al., 2014).

Empirical software engineering is no different (Da Silva et al., 2014). In the context of IaC, which is an under-explored research topic within software engineering (Rahman et al., 2018b), replication studies similar to ours can establish the scientific foundations required for secure IaC script development. The need for replication studies is further necessary because of the varying technologies used to implement the practice of IaC (Rahman et al., 2018b). As a hypothetical example, if a research study only focuses on Puppet, then it will have little to no relevance to Ansible users, as an organization that is using Ansible already may not be interested in learning and using research products that are focused solely on Puppet. Furthermore, the IaC technology usage trend is changing: as shown in Figure 2, once Puppet was popular; however, in recent times, Ansible is more popular than Puppet (Rahman et al., 2021b). As such, empirical research needs to keep up with this change in trends. Therefore, a replication of prior work (Rahman and Parnin, 2023) is not only necessary from a theoretical standpoint with respect to generating new knowledge but also from a practical standpoint, to respond to the changing IaC-related trends.

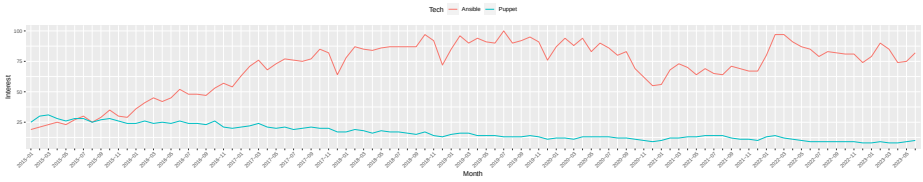

Fig. 2: Interest trends for Ansible and Puppet as determined by Google Search data.

We take motivation from Rahman and Parnin (2023) in our paper, where we study the frequency of propagation, derive a taxonomy of task infections, and obtain practitioner feedback for Ansible scripts. Unlike our TSE paper, we also quantify the relationship between task infection metrics and source code metrics, as well as development activity metrics. Furthermore, to conduct our empirical study, we develop `TIDAL`, as TaintPup from prior research will not be applicable to Ansible. TaintPup can only parse Puppet scripts using the 'puppet parser dump' utility (Labs, 2021), which can not be applied to Ansible. Furthermore, with respect to state reconciliation process and syntax, there are differences between Ansible and Puppet that necessitates differences in understanding of how security weaknesses propagate into code snippets used for state reconciliation. This understanding eventually informs the development process of static analysis tools to detect task infections.

The above-mentioned discussion showcases the progressive nature of our research, where we find gaps in existing research and address these gaps in our paper. The differences between this paper and prior work from Rahman et al. (Rahman and Parnin, 2023; Rahman et al., 2021b) can be further summarized with Table 1.

Table 1: Differences between this paper and recent related work

Characteristic	This paper	TSE 2023 (Rahman and Parnin, 2023)	TOSEM 2021 (Rahman et al., 2021b)
Dataset Timestamp	06/2022	10/2021	11/2018
Development Activity Metrics	Yes	No	No
Impacted Task Taxonomy	Yes	No	No
Propagation Frequency	Yes	Yes (Puppet)	No
Propagation Perception	Yes	Yes (Puppet)	No
Task-based Data Flow Analysis	Yes	No	No
Variable-aware Parsing for Ansible Scripts	Yes	No	No

Table 1 shows that as part of our replication study, we have investigated task-based flow analysis, the impact of development activity metrics, characterized impacted tasks, and constructed a tool that performs variable-aware parsing of Ansible scripts. All of these contributions expand the science of IaC-based infrastructure management, which can be further leveraged for future quality assurance research in the domain of IaC. Development activity metrics have been used as indicators of quality concerns for IaC in prior work (Rahman et al., 2020b). From a practitioner standpoint, the implication using these indicators is prioritizing quality assurance efforts, such as inspection and testing. Prior research on IaC have not investigated the correlation between security weakness presence and development activity metrics. We address this limitation by quantifying the correlation between development activity metrics and task infection presence. Identified development activity metrics can later be used to prioritize inspection efforts for Ansible scripts for which task infection appears.

There are similarities between our paper and the study we replicated (Rahman and Parnin, 2023). We have identified infrastructure categories that appear for both Ansible and Puppet: 'continuous integration', 'data storage', 'networking', and 'virtualization'. Similar to prior work (Rahman and Parnin, 2023), surveyed practitioners from our study perceive data storage to have the most severe impact if affected by security weaknesses. These insights strengthen the empirical science of security weakness propagation as they demonstrate that (i) yes, security weaknesses do propagate into provisioned infrastructure, and (ii) data storage-related infrastructure is perceived to bear the most severe consequences if affected by security weaknesses.

Along with confirmation of existing evidence in the context of Puppet, our replication study also revealed new insights that are different from the study that we have replicated. For example,

- We have identified new categories of impacted infrastructure, namely, 'antivirus' and 'message broker'.
- We find the most frequently occurring category to be 'data storage' for Ansible scripts, whereas the most frequently occurring category for Puppet is 'continuous integration'.

– Unlike Puppet scripts, we do not observe security weaknesses propagating into infrastructure related to load balancers and communication platforms.

Therefore, our empirical study has derived results that are different from the our original study Rahman and Parnin (2023). Shull et al. (2008) stated "*a replication that produces results different from those of the original experiment can also be viewed as successful*". In short, our empirical study provides support for Carver et al. (2010)'s observations: "*one of the main benefits of an experimental replication is that it provides to researchers the ability to confirm, refute, or deepen the conclusions drawn from an earlier study.*"

A brief conclusion of the above-mentioned discussion is:

– This paper is the first to quantify task infection frequency in Ansible scripts;
– This paper has identified two new categories related to impacted infrastructure, namely, 'antivirus' and 'message broker';
– This paper has proposed an automated technique to detect task infections by incorporating variable-aware parsing and data flow analysis;
– This paper is the first study to investigate the relationship between task infections and development activity metrics;
– This paper shows that for Ansible scripts, the most frequently occurring category related to impacted infrastructure is 'data storage' for Ansible scripts, whereas the most frequently occurring category for Puppet scripts is 'continuous integration';
– Unlike Puppet scripts, we do not observe security weaknesses to propagate into infrastructure related to load balancers and communication platforms in the case of Ansible scripts;
– This paper confirms the findings from Rahman and Parnin (2023): infrastructure categories that appear for both Ansible and Puppet scripts are: 'continuous integration', 'data storage', 'networking', and 'virtualization'; and
– This paper confirms survey-related findings from Rahman and Parnin (2023) who observed data storage to have the most severe impact if affected by security weaknesses.

We organize the rest of the paper as follows: we provide the methodology of our empirical study in Section 2 with the necessary background. Then, we describe our empirical findings in Section 3. Next, we discuss the implications of the paper in Section 4 followed by a discussion of threats in Section 5. We then briefly describe related work in Section 6. Finally, we conclude the paper in Section 7.

2 Methodology

We describe the methodology of our empirical study in this section. An overview of our methodology is presented in Figure 3.

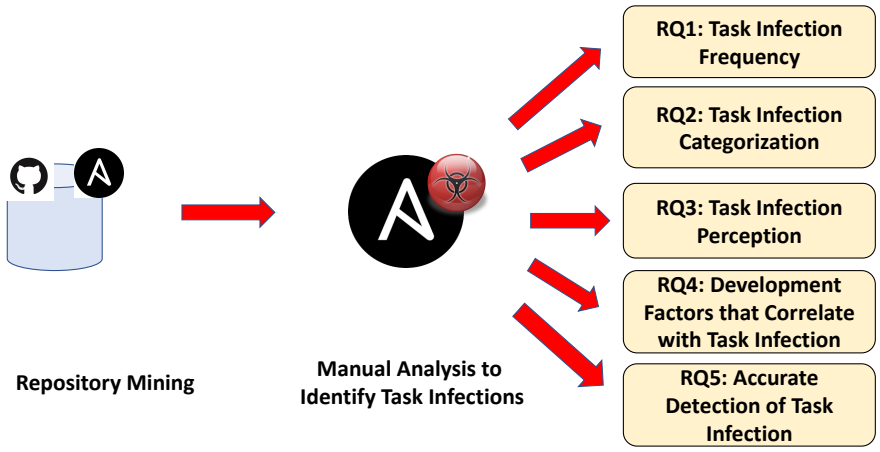

Fig. 3: An overview of our methodology.

2.1 Methodology for RQ1

We answer RQ1 by describing our process of determining task infections in our set of 27,213 scripts by first providing the necessary background.

2.1.1 Background

Background on Ansible Scripts: Ansible is a popular tool to implement the practice of IaC (Rahman et al., 2021b). In Ansible, computing infrastructure is managed with scripts written in YAML, which are sometimes colloquially referred to as manifests. Following recent Ansible-related research (Rahman et al., 2021b; Saavedra and Ferreira, 2023; Opdebeeck et al., 2023), we use the term 'scripts' throughout the paper. We use Listing 1 to demonstrate the contents of an example Ansible script. Using the `hosts: all` keyword, the script specifies that all tasks specified in the script will apply to all applicable hosts. The `tasks` keyword is used to list all the tasks executed as part of the script. The script specifies two tasks namely, `Add test user 'test-user'` and `Copy file content`. To perform these tasks the script leverages two built-in Ansible libraries namely, `ansible.builtin.user` and `ansible.builtin.copy`. A collection of tasks can be executed as a 'play'. An Ansible script with one or more plays is called an Ansible playbook. `Tasks` can be automatically loaded using `roles`. One `role` can refer to multiple `tasks` in order to reuse and share Ansible code efficiently.

In the case of Ansible scripts, the identity of required computing infrastructure is specified using a `hosts` file developed in YAML or INI. With `hosts: all`, Ansible executes all tasks specified in the script for all computing infrastructure specified by the `hosts` file. Listing 2 provides an example `hosts` file. The file includes the IP addresses and necessary authorization information for

```yaml
---
- name: An Example Ansible Script
  hosts: all

  tasks:
  - name: Add test user 'test-user'
    ansible.builtin.user:
         name: test-user
         comment: A test user
         uid: 0007
  - name: Copy file content
    ansible.builtin.copy:
         src: /tmp/myfiles/example.txt
         dest: /home/example.txt
         owner: test-user
```

Listing 1: An example Ansible script.

one local and two remote computing infrastructure for which both tasks in Listing 1 will be executed. The command to execute the script along with the `hosts` file is `ansible-playbook -v example.yaml -i hosts`.

Code snippets in Listing 1 and 2 showcase how tasks are pivotal to setup and manage computing infrastructure. As part of our empirical study, we focus on tracking the flow of a security weakness into a task, which we define as task infection.

```
[server1]
3.230.71.200
ansible_ssh_user=ubuntu
ansible_ssh_private_key_file=<HIDDEN>/aws-private.pem

[server2]
10.230.71.300
ansible_ssh_user=ubuntu
ansible_ssh_private_key_file=<HIDDEN>/aws-private.pem

[server3]
19.20.71.250
ansible_ssh_user=ubuntu
ansible_ssh_private_key_file=<HIDDEN>/aws-private.pem
```


Listing 2: An example `hosts` file needed to execute the tasks listed in Listing 1.

Background on Ansible's State-based Approach: Ansible uses a state-based approach to manage computing infrastructure (Ansible, 2020). Ansible uses a code element called 'task', which is used to perform necessary changes to the relevant computing infrastructure (Opdebeeck et al., 2022). While executing a task, Ansible first queries about the state of the infrastructure, i.e., if the necessary configurations specified in a task exist in the infrastructure (Opde-

beeck et al., 2022). Ansible only makes changes if the state of the infrastructure is different from the configurations specified in the task. Let us consider Listing 3, which shows a task to create a text file called 'example.txt'. While executing the task, Ansible will first query the computing instance of interest to determine if the 'example.txt' file exists with three keys: 'path', 'mode', and 'owner'. Each of these keys corresponds to a configuration for the file 'example.txt'. The path, mode, and owner are expected to be, respectively, '/temp', '0755', and 'sample'. If 'example.txt' exists with the provided configurations, then Ansible will not make any changes to the instance. Ansible will make changes to the instance if the file does not exist or if the file exists, but the configurations are not present.

```
- hosts: all
  tasks:
    #Task to create a file
    - name: Create example.txt
      file:
        path: /temp
        mode: 0755
        owner: test
```

Listing 3: An example task to create 'example.txt' with the following configurations: path, mode, and owner.

A security weakness is an insecure coding pattern used by a code element. In the context of Ansible scripts security weaknesses are recurring coding patterns that are used by a task. As configurations specified in tasks determine the desired configurations of the computing infrastructure, security weaknesses that propagate in tasks will impact Ansible-based management of computing infrastructures. We describe the phenomenon of propagation of security weaknesses into a task as task infection. According to our definition if at least one security weakness propagates into a task then we identify that task to be infected. A task into which at least one security weakness propagates is defined as an infected task. One or multiple security weaknesses can propagate into multiple tasks.

2.1.2 Dataset Construction

We use OSS repositories hosted on GitHub. As OSS repositories are susceptible to quality concerns, such as including personal projects, we use an existing dataset curated by Hassan and Rahman (2022). Hassan and Rahman (2022) systematically applied a set of filtering criteria where they used developer count, the existence of Ansible tasks, and repository maturity to curate Ansible repositories mined from GitHub. From their dataset, we identify 56 repositories. A breakdown of how the filtering criteria was applied to get the set of

104 repositories is available in Table 2. Attributes of the repositories are available in Table 3. We observe that the count of tasks mined from our repositories is 49,898.

Hassan and Rahman (2022) applied the following criteria to obtain the set of 104 repositories:

- *Criterion-1*: The repository includes Ansible scripts.
- *Criterion-2*: The repository is not a clone of another repository.
- *Criterion-3*: Count of developers is at least five.
- *Criterion-4*: The repository has at least two commits per month.
- *Criterion-5*: The lifetime of the repository is at least one month.
- *Criterion-6*: Similar to prior work on IaC Rahman et al. (2019, 2021b, 2020a), the proportion of Ansible scripts is at least 10%.
- *Criterion-7*: The repository must include at least one Ansible test script developed in YAML.

We added another criterion along with these seven criteria where we filter repositories with no active issues as of October 01, 2022. Use of issues is a recognized criterion to identify active repositories (Agrawal et al., 2018; Krishna et al., 2018; Rahman et al., 2018a) The dataset provided by Hassan and Rahman (2022) was mined in November, 2020. Our assumption is that some of the repositories may have become inactive, and thus we used this additional criterion to filter such inactive repositories.

Table 2: Repository Filtering

Initial Count	3,405,303
Criterion-1 (Ansible Usage)	6,633
Criterion-2 (Not a Clone)	4,147
Criterion-3 (Contributor Count>3)	856
Criterion-4 (Commits/Month >=2)	770
Criterion-5 (Lifetime>1 month)	675
Criterion-6 (10% Ansible Scripts)	324
Criterion-7 (Existence of Ansible Test Scripts)	104
Criterion-8 (Active issue count > 0)	56

Table 3: Attributes of Dataset

Category	**Count**
Repositories	56
Commits	264,478
Scripts	27,213
Scripts with Tasks	10,374
Total Lines of Code of Scripts	2,860,222
Distinct Tasks	49,898
Contributors	17,446

2.1.3 Dataset Labeling

We use a first-year PhD student as a rater to label the mined 27,213 Ansible scripts from Section 2.1.2. The PhD student has 2 years of professional experience in software engineering. The rater applies closed coding (Saldaña, 2015) where each script is mapped to one or multiple security weakness categories. The rater is provided with the definitions and examples for each security weakness category, as well as the Ansible documentation to perform labeling. Rahman et al. (2021b)'s paper contains examples and a description of the security weakness categories, whereas the Ansible documentation provides details on how configuration values are assigned and used in Ansible tasks.

Rahman et al. (2021b) provided six categories of security weaknesses: empty password, hard-coded secret, no integrity check, suspicious comment, unrestricted IP address, and use of HTTP without SSL/TLS. As part of the dataset labeling activity, first, the PhD student applies pattern matching to identify potential security weaknesses that belong to five categories: empty password, hard-coded secret, use of HTTP without TLS, unrestricted IP address binding, and no integrity check. We do not include suspicious comments as this category is only applicable for comments in Ansible scripts and has no way to propagate into tasks. For pattern matching the rater uses the patterns provided by Rahman et al. (2021b) in their paper to identify security weaknesses. Upon completing this pattern-matching approach, the rater identifies 4,490 security weaknesses. A breakdown of these security weaknesses based on categories is available in Table 4.

Table 4: Count of Security Weaknesses Identified with Pattern Matching

Category	**Count**
Empty password	1
Hard-coded secret	3,199
No integrity check	9
Unrestricted IP address binding	482
Use of HTTP without TLS	799
Total	4,490

As pattern matching is prone to generating false positives the rater manually inspected 1,802 scripts in which at least one security weakness appears. While performing this inspection, the rater determined if the identified security weakness fits the definition of the corresponding category and if the identified security weakness is used within a script of a repository. If a security weakness identified from pattern matching fits the above-mentioned criteria then it is labeled as a true positive, i.e., a true positive instance of task infection. Altogether, this manual activity took 225 hours to complete, where the rater on average spent 7.5 minutes for one script.

The rater identifies 2,621 security weaknesses, as shown in Table 5. We observe a reduction in security weaknesses for each category. Furthermore, the only instance of an empty password is not used by a task and therefore is not

included as a true positive instance. The student is not involved in the design and construction of `TIDAL`. The student was informed by the first author that the dataset will be eventually used to evaluate a tool called `TIDAL`. Upon construction of `TIDAL`, we use the dataset to calculate its accuracy.

Table 5: Count of Security Weaknesses in Our Dataset

Category	**Count**
Hard-coded secret	2,221
Use of HTTP without TLS	261
Unrestricted IP address binding	132
No integrity check	7
Total	2,621

Rater Verification: The rater is the paper's second author who might introduce bias while labeling the scripts. We mitigate this limitation by allocating another rater who is not an author of this paper. The voluntary rater is a third-year Ph.D. student in the department with three years of experience in cybersecurity. The voluntary rater labels 400 randomly-selected Ansible scripts. We randomly select 400 scripts to achieve a >= 95% confidence interval from 27,213 Ansible scripts with a population proportion of 50%. While labeling, both the PhD student and the volunteer who performed rater verification inspected two items: (i) if an identified security weakness through pattern matching fits the definition of a security weakness category, and (ii) if the identified security weakness propagates into one task. Therefore, the raters inspect each task in each script to determine the presence of task infection.

We have computed the Cohen's Kappa between the PhD student and the voluntary rater for all task infections documented in the set of 400 Ansible scripts. A breakdown of Cohen's Kappa with interpretation based on Landis and Koch's observations (1977) is available in Table 6. We observe the overall agreement between the PhD student and the voluntary rater to be 'substantial', but it varies from one category to another because of disagreements. For example, the voluntary rater disagreed with an instance of task infection for a hard-coded secret, as the hard-coded username is used for setting up a user ID, which the voluntary rater felt was incorrect. The ratings of the PhD student and the voluntary rater is available in our replication package (Rahman, 2023).

Table 6: Cohen's Kappa for Rater Verification

Category	**Kappa**	**Interpretation**
Hard-coded secret	0.65	'Substantial'
Insecure HTTP	0.63	'Substantial'
Invalid IP address	1.00	'Perfect'
None	0.67	'Substantial'
No integrity check	0.40	'Fair'
All	0.66	'Substantial'

We also use another rater who do not participate any rater verification to identify instances of false negatives. The rater inspected a set of 50 Ansible scripts not used during rater verification to identify false negatives. The rater did not identify any false negatives.

In their paper, Rahman et al. (2021b) reported 194 instances of missing integrity checks, whereas we reported 7 instances. Possible explanations include (i) differences in datasets as we do not use the same set of Ansible scripts similar to Rahman et al. (2021b); (ii) potential problematic implementations of rules as acknowledged by Rahman et al. (2021b) themselves; and (iii) not accounting for data flow analysis. We include instances of missing integrity checks if they are used by a task.

2.1.4 Quantifying Task Frequency

We answer RQ1 with the identified task infections as determined from our manual analysis. We compute three metrics: *first*, the count of tasks in each Ansible script in which at least one security weakness propagates. *Second*, we calculate the proportion of tasks with a metric called 'Infected Task (%)' using Equation 1. The 'Infected Task' metric provides a quantitative summary on average of how frequently security weaknesses propagate into tasks. *Third*, we compute 'Script-wise Task Infection (%)' using Equation 2. This metric computes the proportion of tasks within a script that are impacted by security weaknesses. For example, in a script, if there are 4 tasks, of which security weaknesses propagate into 2, then the Script-wise Task Infection (%) value will be 50%. We repeat the calculation for all scripts with at least one task infection.

$$\text{Infected Task}(\%) = \frac{\text{\# of tasks in which} >= 1 \text{ security weaknesses propagate}}{\text{total \# of unique tasks in the dataset}} * 100\% \quad (1)$$

$$\text{Script-wise Task Infection } (\%) = \frac{\text{\# of tasks within a script in which} >= 1 \text{ security weaknesses propagate}}{\text{total \# of unique tasks in a script with} >= 1 \text{ task infection}} * 100\% \quad (2)$$

2.2 Methodology for RQ2

We answer RQ2 by identifying tasks into which security weaknesses propagate. The `name` key is used to describe the changes that are made to computing infrastructures. Then, we apply open coding with the text obtained from the values of the `name` key. A `name` is not mandatory for a task, and therefore, if a task is missing a name, then we use the text content within the task and use that content in our open coding analysis. Open coding is a qualitative

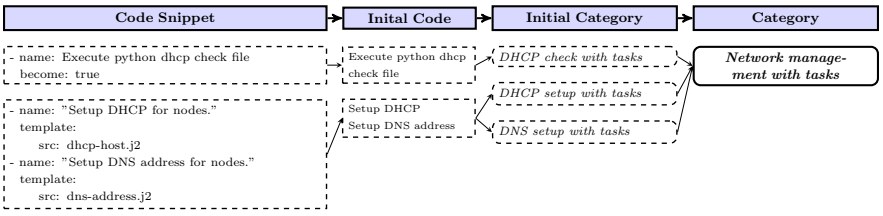

Fig. 4: An example to demonstrate our open coding process to derive categories.

analysis technique that groups similar text into categories (Saldaña, 2015). We illustrate our open coding process using Figure 4. First from the code snippets of tasks we obtain initial codes. For each initial code, we identify the initial category. For example from the initial code 'Execute python dhcp check file', we derive the initial category 'DHCP check with tasks' as the code corresponds to using a Python file to check for DHCP, i.e., the dynamic host configuration protocol, a network management protocol used on Internet Protocol networks to automatically assign IP addresses (Droms, 1999). Finally, we merge the three initial categories into one category called 'Network management with tasks' as all initial categories indicate network-related management with Ansible tasks.

In our categorization one task can belong to multiple categories. Figure 5 shows one task to use a hard-coded username to manage authorization for Jenkins and MySQL, tools respectively, used for continuous integration and data storage. After deriving the categories, we quantify the proportion of tasks that belong to specific categories using Equation 3.

$$\text{Category-wise Infection (x, y\%)} = \frac{\#\text{ of } y\text{-related tasks in which } >= 1 \text{ weaknesses of category } x \text{ propagate}}{\text{total } \#\text{ of tasks in which } >= 1 \text{ weaknesses of category } x \text{ propagate}} * 100\% \quad (3)$$

Rater Verification: The open coding process is conducted by the first author making the open coding process susceptible to rater bias. We mitigate this limitation by using another rater who is the last author of the paper. We randomly select 350 tasks to achieve a >= 95% confidence interval from the identified 1,805 tasks with a population proportion of 50%. As detailed in Section 3.1, we obtain 1,805 tasks that are affected by >= 1 security weaknesses. For the set of 350 tasks we observe a Cohen's Kappa (Cohen, 1960) of 0.82, indicating 'almost perfect' agreement (Landis and Koch, 1977).

```
sample_user: "root" <-------------- Hard-coded username
sample_group: "root"
mysql_root_password: "root" <-------------- Hard-coded password
...
# The Jenkins account needs a login shell because
    ↪ Jenkins uses scp
- name: Add the root user for Jenkins and MySQL setup
            user: name={{ jenkins_user }} append=yes
               ↪ group={{ jenkins_group }}
               ↪ shell=/bin/bash
            shell: >
            mysql -u root -NBe
            'CREATE USER "{{ sample_user }}"@"{{ item
               ↪ }}" IDENTIFIED WITH
               ↪ mysql_native_password BY "{{
               ↪ mysql_root_password }}";'
            with_items: "{{
               ↪ mysql_root_hosts.stdout_lines|default([])
               ↪ }}"
            when: ((mysql_install_packages | bool) or
               ↪ mysql_root_password_update) and
               ↪ ('5.7.' in mysql_cli_version.stdout)
```

Fig. 5: An example where one task performs two types of infrastructure management tasks: continuous integration and data storage.

2.3 Methodology for RQ3

We answer RQ3 by conducting an online survey with practitioners who develop Ansible scripts. We develop the survey by first asking practitioners about their experience in Ansible script development. Next, we briefly describe the identified task categories from Section 2.1.4. Then, we ask one question about frequency and one question about severity. In particular, we ask "*At what frequency do you think the task categories listed below can be infected by security weaknesses?* ". Next, we ask "*What is the severity of the task categories listed below?*". We follow Kitchenham and Pfleeger's guidelines (Kitchenham and Pfleeger, 2008) and explain the preservation of confidentiality of survey respondents, describe the expected amount of time it might take to complete the survey, provide explanations of the purpose of the study, and use five-item Likert questions. We use a Likert scale to quantify perception because for personal opinion surveys where practitioners are asked to express their opinions, the Likert scale is recommended (Kitchenham and Pfleeger, 2008).

The survey questionnaire is available in Table 7. The motivation for SQ1 in the survey is to quantify the experience of survey respondents in using Ansible. The motivation for SQ2 is to compare practitioner perceptions with empirical data obtained for RQ1 and identify similarities and differences. Observed similarities will further substantiate empirical data, whereas differences will provide nuanced perspectives on Ansible script development. The motivation for SQ3 is to gain an understanding of what type of impacted tasks are severe as per practitioner perceptions. Generated findings from this research question will advance the science of security for Ansible and also lay

the groundwork for further research investigations. For example, if practitioners perceive data storage-related tasks to be impacted more severely than others, then researchers can apply empirical analysis to support or refute their perceptions.

Table 7: Questions Asked in the Survey

Index	Question Text
SQ1	How long have you used Ansible in a professional setting? [Options: 1-2 years, 3-4 years, 4-5 years, and > 5 years.]
SQ2	At what frequency do you think the task categories listed below can be infected by security weaknesses? [Options: Not at all frequent, rarely frequent, somewhat frequent, frequent, highly frequent.] The categories are: continuous integration (CI), virtualization (VIRT), antivirus (ANTI_VIRUS), message broker (BROKER), data storage (STORAGE), and networking (NETWORK).
SQ3	What is the severity of the task categories listed below? [Options: Not at all severe, low severity, moderate severity, severe, highly severe.] The categories are: continuous integration (CI), virtualization (VIRT), antivirus (ANTI_VIRUS), message broker (BROKER), data storage (STORAGE), and networking (NETWORK).

We deploy the survey to practitioners via email. We sent out emails to 250 practitioners, whose emails we collected by mining the 56 repositories obtained from Section 2.1.2. Following Smith et al. (2013)'s guidelines, we offer a drawing of one 25 USD Amazon gift card as an incentive to participate in the survey. We deploy the survey from March 2022 to September 2022. Prior to deploying the survey, we obtain Internal Review Board (IRB) approval (IRB #2356). According to our IRB approval process, we:

- Do not release any private and sensitive information of the survey participants;
- Do not commercially advertise any existing IaC-related research of the research group;
- Do not use automation to send emails. We send email messages where we made it clear that the purpose of the email is only to seek feedback on our research;
- Explicitly mention that participation or lack thereof will not impact their occupation;
- Provide full identity of the lead researcher who is conducting the survey; and
- Seek approval from each participant via emails prior to sending the survey.

We apply Chi Squared test (Greenwood and Nikulin, 1996) to compare if practitioner perception is significantly different for the task infection categories. The null and alternative hypotheses are:

- Null: There is no difference between the task infection categories with respect to practitioner perceptions.

– Alternate: There are differences between the task infection categories with respect to practitioner perceptions.

2.4 Methodology for RQ4

With RQ4, we aim to identify development factors that correlate with the presence of task infections in Ansible scripts. Such correlation can help give practitioners insights into managing their Ansible development process. We answer RQ5 by first mining metrics and then by applying logistic regression (Long and Freese, 2006) as discussed in the following subsections:

2.4.1 Mining Metrics

We answer RQ4 using two categories of metrics:

– Source code metrics: Metrics that are computed using the source code of each Ansible script. In this case, we rely on prior work from Palma et al. (2020) who have provided a list of 46 metrics that are related to quality aspects of Ansible scripts. We do not use the metrics provided by Rahman and Williams (2019) as these metrics are applicable for Puppet scripts, and not Ansible scripts. The name and definition of each source code metric is provided in Table 8.
– Development activity-related metrics: Metrics that are computed by synthesizing development activity patterns for each Ansible script. The name and definition of each development activity-related metric is provided in Table 8. This category of metrics can be further divided into two groups:
  – Generic metrics: We leverage prior work that has quantified relationship between metrics related to development factors and quality for generic software projects. We use the following metrics that are applicable for generic software projects: age, commits, scatteredness, and minor contributors. These metrics are reported in existing publications (Hassan, 2009; Rahman and Devanbu, 2013; Nagappan and Ball, 2005; Bird et al., 2011; Radjenović et al., 2013) that have quantified the correlation between development activity metrics and software quality.
  – Ansible-related: We also use metrics that are unique to Ansible. We derive these metrics by analyzing Internet artifacts, such as blog posts that have discussed development factors that are correlated with quality of Ansible scripts. We use Internet artifacts as practitioners often express their opinions in Internet artifacts instead of peer-reviewed publications, such as conference and journal publications.
  We analyze Internet artifacts by *first* curating Google search results obtained from the search string 'quality development of Ansible scripts'. As part of this curation process, we collect the top 100 search results, and exclude results that are not related to Ansible script development. We obtain 11 artifacts from the set of 100 search results upon completion of the curation process. *Second*, from the curated search results

we read each artifact to identify if a development factor is explicitly mentioned to be related to quality. We identify two development factors namely, use of BASH commands instead of Ansible modules, and not using Ansible roles to be related to quality Ansible script development. As part of our analysis, we use two metrics called 'Bash Envy' and 'IsRole' that respectively, correspond to use of BASH commands instead of Ansible modules, and not using Ansible roles to be related to quality.

Altogether, we use 6 development activity-related metrics and 44 source code-related metrics to answer RQ4. A list of source code-based metrics and development activity-based metrics is provided respectively, in Table 8 and Table 9 with names and definitions.

$$k_i = \frac{\text{number of times line } i \text{ is modified}}{\text{number of commits for the script}} \tag{4}$$

$$Scatteredness = -\sum_{i=1}^{N}(k_i log_2 k_i) \tag{5}$$

Table 8: Source Code Metrics Used to Answer RQ4

Name	Definition
Average Play Size	Lines of source code in playbooks divided by count of plays
Average Task Size	Lines of source code in tasks divided by count of tasks
Blocks	Count of *block* syntax occurrences
Commands	Count of *command*, *expect*, *psexec*, *raw*, *script*, *shell*, and *telnet* syntax occurrences
Conditions	Count of *is*, *in*, ==, ! =, >, >=, <, <= occurrences in *when*
Decisions	Count of *and*, *or*, *not* syntax occurrences in *when*
Deprecated Keywords	Count the occurrences of deprecated keywords
Deprecated Modules	Count the occurrences of deprecated modules
Distinct Modules	Count of distinct modules maintained by the community
Ensure count	Count of "$+.stat.+isdefined$" regex matches in *when*
Error Handling Blocks	Count of block-rescue-always section occurrences
Error Ignores	Count of *ignore_errors* syntax occurrences
External Modules	Count occurrences of modules not maintained by the community
Fact Modules	Count occurrences of fact modules
Files	Count of *file* syntax occurrences
File Mode	Count of *mode* syntax occurrences
Filters	Count of — syntax occurrences inside $*$ expressions
Includes	Count of *include* syntax occurrences
Keys	Count of keys in the dictionary representing a playbook or tasks
Lines of code	Count of lines in a script
Lines of blank	Count of empty lines in a script
Lines of comments	Count of statements that are comments
Lookups	Count of *lookup*($*$) occurrences
Loops	Count of *loop* and *with_** syntax occurrences
Math Ops	Count of +, −, /, //, %, $*$, $**$ syntax occurrences
name with variables	Count of *name* occurrences matching the "$.*+.*$" regex
Parameters	Count the keys of the dictionary representing a module
Paths	Count of *paths*, *src* and *dest* syntax occurrences
Playbook Imports	Count of *import_playbook* syntax occurrences
Plays	Count of *hosts* syntax occurrences
Regexes	Count of *regexp* syntax occurrences
Role Imports	Count of *import_role* syntax occurrences
Role Includes	Count of *include_role* syntax occurrences
Role Size	Total length of the *roles* section
SSH Authorized Keys	Count of *ssh_authorized_key* syntax occurrences
Susp. Comments	Count comments with *TODO*, *FIXME*, *HACK*, *XXX*, *CHECKME*, *DOCME*, *TESTME*, or *PENDING*
Task Imports	Count of *import_tasks* syntax occurrences
Task Includes	Count of *include_tasks* syntax occurrences
Task Size	Total length of the all *tasks*
Tokens	Count the words separated by a blank space
Text Entropy	Complexity of the script based on information content
Unique Names	Count of *name* syntax occurrences with unique values
URLs	Count of *url* syntax occurrences
User Interactions	Count of *prompt* syntax occurrences
Var Includes	Count of *include_vars* syntax occurrences
var Length	Total length of all *vars* in all plays

Table 9: Development Activity Metrics Used to Answer RQ4

Name	Definition
Age	Age of a script as measured by the difference between the last commit timestamp and first author timestamp. Prior research (Radjenović et al., 2013) has shown age to be correlated with software defects. Therefore, we hypothesize that the probability of task infection presence is higher for older scripts. We compute this metric by measuring the differences between the two timestamps and then converting the difference into days.
BASH Envy	Count of instances when the `command` or the `shell` module is used to execute a BASH or BAT command. Practitioners perceive the use of `command` or `shell` as a bad practice, which may introduce defects (Davis, 2019). We hypothesize that the probability of task infection presence is higher for scripts that have higher BASH Envy values.
Commits	Count of commits made for a script. Prior research (Nagappan and Ball, 2005; Rahman and Devanbu, 2013) has shown commits to correlate with the presence of software defects. We hypothesize that the probability of task infection presence is higher for scripts that have higher commits.
isRole	This metric determines if a script is part of a `role`. `Roles` provide a mechanism to group similar `task` properties, such as `handlers` and `vars` to increase reusability (Ansible, 2020). If the script is part of a `role`, we assign '1', and '0' if otherwise. Practitioners recommend use of `roles` as a good practice Ansible (2022); Cozens (2022), which can mitigate defects in Ansible scripts. We hypothesize the probability of task infection presence is higher for scripts that are not part of a role.
Minor contributors	Count of developers who modify $< 5\%$ of the total lines of code for a script. The 5% threshold is provided by Bird et al. (2011) in their paper titled 'Don't Touch My Code! Examining the Effects of Ownership on Software Quality'. Prior research (Rahman and Devanbu, 2013) has shown a minor contributor count to correlate with the presence of software defects. We hypothesize the probability of task infection presence is higher for scripts that have higher minor contributors.
Scatteredness	This metric computes if submitted code changes for a script are spread out across the script or should be grouped together in a specific location of a script. Based on findings from Hassan (Hassan, 2009), we hypothesize that the probability of task infection presence is higher for scripts that have higher scatteredness. In Equation 4, we calculate k_i, which we use in Equation 5 to quantify the scatteredness of a script. For example, let us assume $Script\#A$ has ten lines of code and six commits. Three modifications are made to lines #6 and 7 each. According to Equation 5, the scatteredness score for $Script\#A$ is 0.8.

2.4.2 Applying Logistic Regression

We use logistic regression (Long and Freese, 2006) to quantify the correlation between the presence of task infection and development metrics as well as the correlation between task infection and source code metrics that we have listed in Section 2.4.1. We do not use individual metric-based statistical test, such as Mann Whitney U test (Mann and Whitney, 1947), as the metrics may have a combined effect on the dependent variable, which we cannot account for by applying Mann Whitney U test. We construct three logistic regression models:

- Development activity-based model: A logistic regression model where all independent variables are development activity-based metrics. The dependent variable is the presence of task infection in an Ansible script. The independent variables are age, BASH Envy, commits, isRole, minor contributors, and scatteredness. Except for 'isRole', all metrics are numeric.
- Source code-based model: A logistic regression model where all independent variables are the 46 source code-based metrics. The dependent variable is the presence of task infection in an Ansible script. All 46 metrics are numeric.
- Combined model: A logistic regression model where all development activity-based and source code-based metrics are included as independent variables. The dependent variable is the presence of task infection in an Ansible script.

In each of the three logistic regression models, the dependent variable is the presence of at least one task infection in an Ansible script. The dependent variable can be any of two possible values: 1 and 0, respectively, indicating the presence and absence of task infections.

Before applying the logistic regression, we apply the following recommended practices: (i) apply log transformation to reduce heteroscedasticity (Cohen et al., 2014); and (ii) test if multi-colinearity exists between the independent variables using variable influence factor (VIF) (Gelman and Hill, 2006), where VIF > 5 is the threshold to determine whether or not a metric exhibits multi-colinearity (Menard, 2002).

For each of our constructed logistic regression models, we report:

- $p - value$ for each independent variable. Following Cramer and Howitt's observations (Cramer and Howitt, 2004), we determine a metric to correlate with the presence of task infection if the $p - value$ for that metric is < 0.01; and
- coefficients, deviance, odds ratio, and the sum of square errors for each independent variable (Hosmer Jr et al., 2013; Long and Freese, 2006).

2.5 Methodology for RQ5: Task Infection Detector for Ansible Scripts (`TIDAL`)

Manual identification of task infections is not practical for a practitioner who develops and manages multiple Ansible scripts. A practitioner may prefer a tool that automatically identifies security weaknesses in Ansible scripts and reports which tasks are affected by security weaknesses. Especially, considering the fact that practitioners seek information on how reported static analysis alerts are used in the code base (Smith et al., 2015), it is pivotal to not only report the security weaknesses but also report which tasks are being impacted by security weaknesses. To that end, we focus on developing an automated technique that can identify task infections.

Table 10: Rules Obtained from Rahman et al. (2021b) to Detect Security Weakness

Weakness Name	Rule
Hard-coded secret	$(isKey(k) \wedge length(k.value)>0) \wedge (isUser(k) \vee isPassword(k) \vee isPrivateKey(k))$
Insecure HTTP	$(isKey(k) \wedge isHTTP(k.value))$
No integrity check	$(isKey(k) \wedge (isIntegrityCheck(x) == False \wedge isDownload(x.value)))$
Unrestricted IP address	$(isKey(k) \wedge isUnrestrictedBind(k.value))$

2.5.1 Step-1: Syntax Analysis

We describe the construction of `TIDAL` by discussing how `TIDAL` identifies security weaknesses by detecting task infections. `TIDAL` leverages Ansible's state-based approach to detect security weaknesses in Ansible scripts. `TIDAL` detects task infections as follows:

`TIDAL` performs syntax analysis using the following steps:

- *Key value pair extraction*: `TIDAL` uses PyYaml [1] to extract keys and their corresponding values as key-value pairs.
- *Identifying keys with hard-coded configuration values*: Upon key-value pair extraction, `TIDAL` identifies keys with values that are hard-coded and with values specified by variables. For example, `password: 'sat_pass'` is a key with a hard-coded configuration value. On the other hand, `password: ''{{ ironic_db_password }}''`, is a key where value is inherited from another variable `ironic_db_password`.

2.5.2 Step-2: Security Weakness Identification

`TIDAL` identifies potential security weaknesses using the keys with hard-coded configuration values. `TIDAL` does not ignore variables that store values in a key-value format. For example, variables, such as `password: '' ironic db password ''` obtain data from another key-value pair defined in another Ansible script. `TIDAL` performs rule matching to identify potential security weaknesses in Section 2.5.1. `TIDAL` uses a set of rules provided by Rahman et al. (2021b) to identify four categories of security weaknesses. The rules are listed in Table 10. The patterns used by each of these rules are listed in Table 11. With the rules in Table 10, `TIDAL` identifies security weaknesses.

2.5.3 Step-3: Data Dependence Graph

`TIDAL` constructs data dependence graphs (DDGs) to detect task infections. In each DDG, two categories of nodes exist a taint node and a sink node. A taint node is a key of key-value pair that is identified as a security weakness

[1] https://pyyaml.org/

Table 11: String Patterns Used for Functions in Rules Presented in Table 10

Function	String Pattern
isDownload()	'http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@.&+]|[!*,]|(?:%[0-9a-fA-F][0-9a-fA-F]))+.[dmg|rpm|tar.gz|tgz|zip|tar]'
isHTTP()	'http:'
isUnrestrictedBind()	'0.0.0.0'
isIntegrityCheck()	'gpgcheck', 'check_sha', 'checksum', 'checksha'
isPassword()	'pwd', 'pass', 'password'
isPrivateKey()	'[pvt|priv]+*[cert|key|rsa|secret|ssl]+'
isUser()	'user'

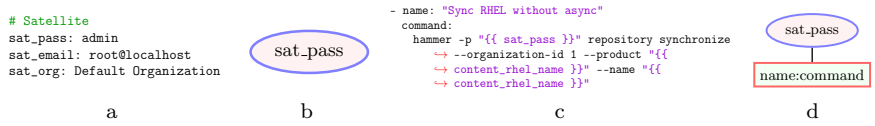

Fig. 6: An example to demonstrate TIDAL's DDG construction process.

from Step 2. A sink node is a key used as part of specifying configurations within a task. An edge exists between a taint node and a sink node if the taint node is reachable from the sink node. A taint node reaches a sink node if the value of the key identified from the taint node is not changed and used by the key identified by the sink node. We use def-chain analysis (Aho et al., 1986) to determine reachability. Upon construction of a DDG, TIDAL queries the DDG to determine if a path exists between the taint and the sink node. If a path exists, TIDAL determines that corresponding security weakness identified by the taint node as a true positive.

Example to demonstrate TIDAL's execution process: We provide a running example to demonstrate the construction of a DDG. In Figure 6a, we observe an Ansible script with a hard-coded password (`sat_pass: admin`). The key-value pair is < `sat_pass`, `admin` >. As shown in Figure 6b, the taint node is `sat_pass`, as `sat_pass` is the key in the identified hard-coded password. From Figure 6c, we observe that `sat_pass` is used in a task with the `command` key: `hammer -p '' sat_pass '' repository synchronize --organization-id 1 --product '' content_rhel_name '' --name '' content_rhel_name ''`. Therefore, according to our DDG construction process `command` is a sink node as shown in Figure 6d. The constructed DDG is shown in Figure 6d.

TIDAL accurately detects an instance of task infection only if both these requirements are satisfied, i.e., a detected pattern is, in fact, a security weakness, and the detected security weakness propagates into a task. TIDAL is capable of detecting two categories of task infections: (i) task infections that occur within a manifest where the location of security weaknesses and tasks are in the same script, and (ii) task infections that occur in different scripts where the security weakness is in one script, which propagates into a task located in a different script. Figure 7 shows an example of a task infection that oc-

```
ironic_db_password: aSecretPassword473z
ironic_user: admin
- name: "Create ironic user in RabbitMQ"
  rabbitmq_user:
    user: "{{ ironic_user }}"
    password: "{{ ironic_db_password }}"
    force: yes
    state: present
    configure_priv: ".*"
    write_priv: ".*"
    read_priv: ".*"
  no_log: true
```

Hard-coded password

Hard-coded username

Fig. 7: An example of a task infection that occurs within a script.

```
port_range_min: 22
port_range_max: 22
remote_ip_prefix: 0.0.0.0/0
```

Invalid IP address

a Invalid IP address in ‘playbooks/defaults/healthchecks-vars.yml’ script

```
- name: Create security group rules
  os_security_group_rule:
    cloud: default
    interface: internal
    verify: "{{ keystone_service_internaluri_insecure
      ↪ | ternary(false, true) }}"
    security_group: "{{ security_group.name }}"
    protocol: "{{ item.protocol }}"
    port_range_min: "{{ port_range_min }}"
    port_range_max: "{{ port_range_max }}"
    remote_ip_prefix: "{{ remote_ip_prefix }}"
    state: present
  with_items: "{{ security_group.rules }}"
```

Use of invalid IP address in task

b Usage of invalid IP address in ‘playbooks/healthcheck-openstack.yml’ script

Fig. 8: An example of how TIDAL can detect task infections where the source of the security weakness is one script, and the task that is being affected is in another script.

curs within a script. Figure 8 shows an example where an invalid IP address is located in a script called ‘playbooks/defaults/healthchecks-vars.yml’. The security weakness propagates into a task called ‘Create security group rules’ in the script ‘playbooks/healthcheck-openstack.yml’.

Our construction process of TIDAL’s similar to recent work published by Opdebeeck et al. (2023). They constructed GASEL that incorporates three properties: Ansible-aware parsing, incorporation of data flow analysis, and control flow analysis that captures variable precedence. Out of these three properties, TIDAL incorporates two: Ansible-aware parsing and incorporation of data flow analysis. TIDAL’s parser incorporates Ansible-aware parsing so that a variable usage, e.g., ‘‘ username ’’ is not treated as a hard-coded secret. SLAC (Rahman et al., 2021b) does not differentiate between a variable usage and a configuration value and generates a false positive by identifying ‘‘

`username ''` as a hard-coded secret. Unlike SLAC, `TIDAL` also applies data flow analysis by constructing DDGs. However, unlike GASEL, `TIDAL` does not incorporate control flow analysis that captures variable precedence. As a result, `TIDAL` is susceptible to identifying task infections that do not occur due to variable precedence.

2.5.4 Step-4: Evaluation of `TIDAL` with Metrics

We use the dataset created from Section 2.1 to evaluate `TIDAL`'s detection accuracy. For evaluating `TIDAL` we use three metrics: precision, recall, and F-measure. Precision refers to the fraction of correctly identified instances among the total identified security weaknesses, as determined by `TIDAL`. Recall refers to the fraction of correctly identified instances retrieved by `TIDAL` over the total amount instances. F-measure is the harmonic mean of precision and recall (Tan et al., 2005).

3 Empirical Findings

We provide answers to our research questions as follows:

3.1 Answer to RQ1

In this section, we answer **RQ1: How frequently do task infections occur in Ansible scripts?** by reporting the frequency of task infections. We use Table 12 to report the frequency of task infections. Altogether, we identify 2,621 security weaknesses to propagate into 1,805 tasks (3.6% of 49,898). The 'Impacted Task (%)' column in Table 12 shows the proportion of tasks into which >= 1 security weaknesses propagate. The 'Script-wise Task Infection (%)' column in Table 12 shows the distribution of task proportion for a single script in which >= 1 security weaknesses propagate. For example, on average, 6.4% of all tasks within a single script are impacted by >= 1 security weaknesses. Altogether, 1,847 Ansible scripts include at least one task into which at least one security weakness propagates.

Table 12: Answer to RQ1: Frequency of Task Infection

Category	**Impacted Task (%)**	**Script-wise Task Infection (%)** (Avg. Std. Dev.)
Hard-coded secret	3.2	(6.1, 16.3)
No integrity check	0.002	(25.0, 0.0)
Unrestricted IP address binding	0.13	(8.9, 16.6)
Use of HTTP without TLS	0.29	(6.7, 20.0)
Combined	3.6	(6.4, 16.9)

In Table 13, we report the minimum, median, maximum, average, and standard deviation of task count for a script into which >= 1 security weakness propagates. We observe that a security weakness can propagate into as many as 76 distinct tasks. The average task per script varies between 0.4 and 1.0 across four categories.

Table 13: Answer to RQ1: Task Frequency (Minimum, Median, Maximum, Average, Std. Dev.)

Category	**Min, Median, Max, Avg., Std. Dev.**
Hard-coded secret	1, 1, 76, 0.6, 3.3
No integrity check	1, 1, 1, 1, 0.0
Unrestricted IP address binding	1, 1, 12, 0.4, 0.8
Use of HTTP without TLS	1, 1, 20, 0.4, 1.9
Total	1,1,76, 0.6, 3.2

Answer to RQ1: We identify 1,805 of 49,898 tasks to experience task infection in our dataset of 27,213 Ansible scripts.

3.2 Answer to RQ2

In this section, we answer **RQ2: What categories of task infections appear for Ansible scripts?** by first describing the resource categories, which are listed in Figure 9. Next, we report the frequency of resource categories. A mapping of initial categories and the obtained categories is available in Table 14.

Table 14: Mapping of Initial Categories and Final Categories

Initial Category	**High-level Category**
Cron job for anti-virus, Setup antivirus user name	Antivirus-related tasks
Authenticate user for Jenkins, Setup Jenkins port, Setup Jenkins protocol	Continuous integration
Manage AWS Elastic Beanstalk, Setup MySQL user account, Authenticate MongoDB user account	Data storage
Check for private keys needed to access RabbitMQ, Monitor RabbitMQ-related traffic	Message broker
DHCP check with tasks, DHCP setup with tasks	Networking
Manage AWS IAM policies, Setup AWS S3 buckets, Manage AWS Virtual Private Controllers, Authenticate users for LXC containers, Manage certificates for container registries, Manage user configurations for Kubernetes	Virtualization

Fig. 9: Task infection categories.

3.2.1 Description of Task Infection Categories

We identify six categories of task infections. We describe each category with examples as follows.

I-***Antivirus***: Tasks used to manage anti-virus software, i.e., software used to detect and remove computer viruses.

Example: In Figure 10, we observe an instance of a hard-coded username to propagate into a task, which is used to manage packages related to setting up antivirus software. The hard-coded username `antivirus_user` specified in line#2 is used to create an account needed to set up a user for the antivirus software.

```
1 antivirus_app_dir: /edx/app/antivirus
2 antivirus_user: "antivirus" <-------------- Hard-coded username
3 ...
4 - name: create antivirus scanner user
5   user: >
6     name="{{ antivirus_user }}"
7     home="{{ antivirus_app_dir }}"
8     createhome=no
9     shell=/bin/false
```


Fig. 10: An example of a hard-coded user name propagating into a task used to setup antivirus.

II-***Continuous integration (CI)***: Tasks used to manage the infrastructure needed to implement the practice of continuous integration (CI), with tools, such as Jenkins (Jenkins, 2022). CI is the practice of integrating code changes by automatically compiling, building, and executing test cases upon submission of code changes (Duvall et al., 2007).

Example: In Figure 11, we observe an instance of a hard-coded username to propagate into a task, which is used to set up Jenkins, a popular CI tool. The hard-coded username `jenkins_user` specified in line#2 is used to create an account needed to set up a Jenkins shell.

```
1 jenkins_home: /home/jenkins
2 jenkins_user: "jenkins" <-------------- Hard-coded username
3 jenkins_group: "jenkins"
4 ...
5 # roles/jenkins_worker/tasks/system.yml
6 # The Jenkins account needs a login shell because
    ↪ Jenkins uses scp
7 - name: Add the jenkins user to the group and
    ↪ configure shell
8             user: name={{ jenkins_user }} append=yes
                ↪ group={{ jenkins_group }}
                ↪ shell=/bin/bash
```

Fig. 11: An example of a hard-coded user name propagating into a task used to set up a Jenkins shell.

III-***Data storage***: Tasks used to manage the software that is used to store data, such as MongoDB and MySQL.

Example: In Figure 12, we observe an instance of a hard-coded username and an instance of a hard-coded password to propagate into a task, which is used to set up a MySQL user account. The hard-coded username and password is respectively, `mysql_root_username` (line #3) and `mysql_root_password` (line #4).

IV-***Message broker***: Tasks used to implement the practice of message broker, which is used to translate messages written in one formal specification as determined by the sender, to another formal specification as determined by the receiver (Banavar et al., 1999).

Example: In Figure 13, we observe an instance of insecure HTTP to be used by an Ansible task (`rabbitmq_pkg_url`) to set up RabbitMQ, a message broker software in line# 7.

V-***Networking***: Tasks used to manage networking-related functionalities, such as setting up and managing firewalls, virtual local area networks (VLANs) and virtual private networks (VPNs).

Example: In Figure 14, we observe an instance of an unrestricted IP address (`rock_mgmt_nets = [''0.0.0.0/0'']`), which is used to set up a firewall in line# 12.

VI-***Virtualization*** : Tasks used to manage virtualization technologies, such as containers and virtual computing clusters.

Example: Figure 15 shows how a hard-coded username and a hard-coded password propagate into three tasks to set up Red Hat Enterprise Linux (RHEL) products for a virtual computing cluster. The hard-coded username is `sat_user` and the hard-coded password is `sat_pass`, which are later used by three tasks: 'Create Sat Tools product', 'Create Sat Tools repo in the product', and 'Sync RHEL via async'. This example shows that a hard-coded secret may

```
# Hard-coded user name and hard-coded password
mysql_root_home: /root                          Hard-coded username
mysql_root_username: root <----------------------'
mysql_root_password: root <-------------Hard-coded password
...
- name: Update MySQL root password for localhost root
    ↪ account (5.7.x).
  shell: >
    mysql -u root -NBe
    'ALTER USER "{{ mysql_root_username }}"@"{{ item
       ↪ }}" IDENTIFIED WITH mysql_native_password BY
       ↪ "{{ mysql_root_password }}";'
  with_items: "{{
     ↪ mysql_root_hosts.stdout_lines|default([]) }}"
  when: ((mysql_install_packages | bool) or
     ↪ mysql_root_password_update) and ('5.7.' in
     ↪ mysql_cli_version.stdout)
```

Fig. 12: An example of a hard-coded user name propagating into a task used to update a MySQL account.

```
rabbitmq_pkg_url:
    ↪ "http://files.edx.org/rrabbitmq-server_3.2.3-1_all.deb" <-----Insecure
rabbitmq_pkg: "rabbitmq-server"                                      HTTP
rabbitmq_package_checksum_sha256:
    ↪ "e3c377e585c123e06c88422248915f32216641d6f7dfab50d124535c8e93010d"
...
- name: fetch the rabbitmq server deb
  get_url: >
    url={{ rabbitmq_pkg_url }}
    dest=/var/tmp/{{ rabbitmq_pkg_url|basename }}
```


Fig. 13: An example of an insecure HTTP propagating into a task to set up RabbitMQ, a message broker.

reside in one Ansible script and later used in another script. The source of the hard-coded user name and hard-coded password is in an Ansible script called 'conf.sat.perf.yaml', whereas the tasks are listed in a different Ansible script called 'rhel-setup.yaml'.

3.2.2 Frequency of Task Infection Categories

The percentage of the task infection categories is listed in Figure 16. Each column adds up to 100%, showing the proportion of tasks infected by a particular security weakness and mapped to a task category. For example, the total count of tasks infected by security weaknesses is 1,805, of which 26.8% are

```
1# Example of invalid IP address
2rock_mgmt_nets: [ "0.0.0.0/0" ] <---------- Invalid IP address
3...
4- name: Configure firewall ports
5  firewalld:
6    port: "{{ item[1].port }}"
7    source: "{{ item[0] }}"
8    permanent: yes
9    state: enabled
10   immediate: yes
11 with_nested:
12   - "{{ rock_mgmt_nets }}"
13   - { port: "22/tcp" }
```

Fig. 14: An example of invalid IP address propagating into a task to set up a firewall daemon.

used for virtualization. As another example, 144 tasks are infected by insecure HTTP instances, of which 43.0% of tasks are related to virtualization.

Our findings show that computing infrastructure managed with Ansible tasks is frequently infected by security weaknesses, making them susceptible to attacks. We observe hard-coded secrets to propagate into all six categories of tasks.

Answer to RQ2: We identify six categories of task infections: antivirus, continuous integration, data storage, message broker, networking, and virtualization.

3.3 Answer to RQ3: Practitioner Perception

In this section, we answer **RQ3: What are the practitioner perceptions of task infection categories for Ansible scripts?** From our survey, we obtain 23 responses in total. The reported experience in Ansible development is provided in Table 15. In Figures 17 and 18, we report practitioner perceptions for the frequency and severity of the identified task infection categories. The x and y-axis present the percentage of survey participants and task infection categories. For example, from Figure 17, we observe 25% of the total survey respondents to identify continuous integration as a task infection category that frequently or highly frequently appears.

From Figure 17, we observe that the survey respondents perceive networking-related tasks as the most frequent task infection category, where 25% of the total participants found network-related infrastructure to occur frequently or highly frequently. The least frequently perceived task category is anti-virus for

```
# Hard-coded secrets specified in an Ansible manifest
    ↪ ('conf.satperf..yaml')
...
sat_version: "6.3"
sat_user: admin
sat_pass: admin
sat_email: root@localhost
sat_org: Default Organization
sat_orglabel: Default_Organization
sat_orgid: 1
...
# Configuration values used in an Ansible manifest
    ↪ with tasks
 - name: "Create Sat Tools product"
   command:
     hammer -u "{{ sat_user }}" -p "{{ sat_pass }}"
         ↪ product create --organization-id "{{
         ↪ sat_orgid }}" --name "{{
         ↪ content_sattools_name }}"
 - name: "Create Sat Tools repo in the product"
   command:
     hammer --username "{{ sat_user }}" --password "{{
         ↪ sat_pass }}" repository create
         ↪ --content-type yum --label "{{
         ↪ content_sattools_label }}" --name "{{
         ↪ content_sattools_name }}" --organization-id
         ↪ "{{ sat_orgid }}" --product "{{
         ↪ content_sattools_name }}" --url "{{
         ↪ content_sattools_url }}"

 - name: "Sync RHEL via async"
   command:
     hammer -u "{{ sat_user }}" -p "{{ sat_pass }}"
         ↪ repository synchronize --organization-id 1
         ↪ --product "{{ content_rhel_name }}" --name
         ↪ "{{ content_rhel_name }}" --async
   when: "sat_repos_sync == 'async'"
```

Hard-coded user name

Hard-coded password

Fig. 15: An example of a hard-coded password and a hard-coded username propagating into a task used to set up RHEL products and tools for a virtual computing cluster.

which only 10% of the total participants found anti-virus-related infrastructure management to be frequent or highly frequent. From our survey analysis, we observe practitioners' perceptions related to frequency to be incongruent with presented data in Section 3.2.2. According to Figure 16, tasks used to manage data storage is the most frequently infected task category.

The purpose of surveying practitioners was to examine to what extent practitioner perceptions compare to that of mined empirical data. In empiri-

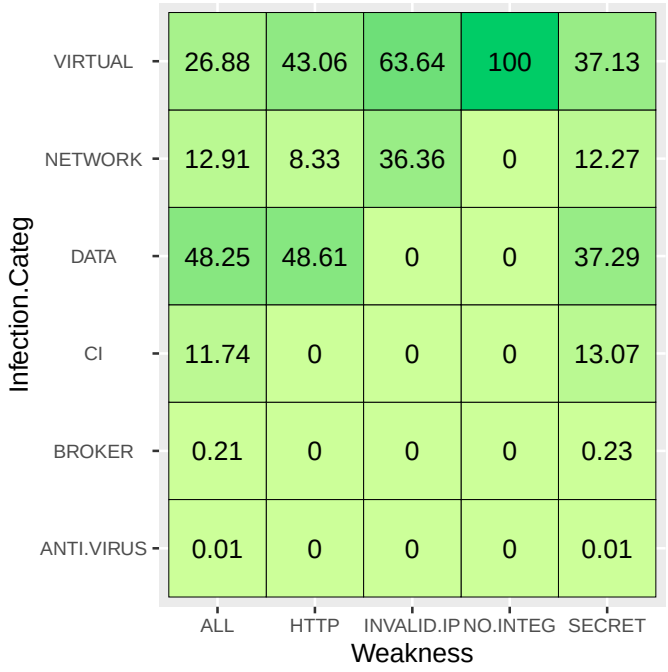

Fig. 16: Answer to RQ2: Percentage of Task Infections for Identified Categories.

Table 15: Survey Respondents' Experience

Experience	Respondent count
$1-2$ years	1
$3-4$ years	5
$4-5$ years	7
>5 years	10

cal software engineering, this is a common approach in mixed methods studies (Easterbrook et al., 2008) The insights that we obtain contribute to the existing body of IaC-related knowledge. We observe that similar to mined data, practitioners also perceive task infection to occur infrequently. Practitioners also find data storage-related tasks to be impacted more severely compared to other categories.

According to Figure 18, we observe 50% of the survey participants to find storage-related tasks to be impacted severely or with high severity. Tasks used to manage CI and networking are perceived to be the second most severe category where 40% of the survey participants found CI and networking to be impacted severely or with high severity.

We also conduct Chi-Squared (χ^2) tests (Greenwood and Nikulin, 1996) for both perceived frequency and perceived severity. In the case of perceived frequency, we observe the χ^2 value to 49.0, with 10 degrees of freedom. The p-

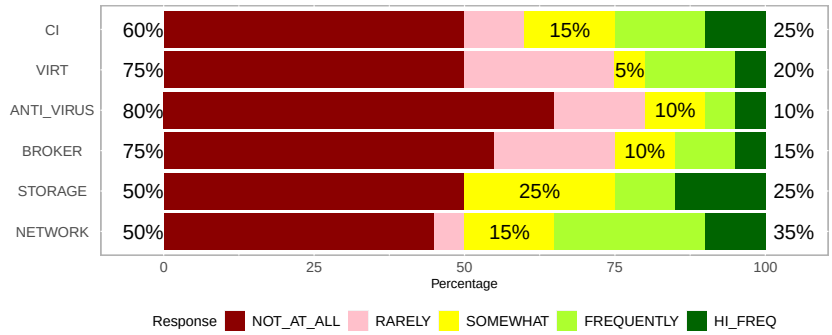

Fig. 17: Answer to RQ4: Perceptions related to the frequency of task infection categories.

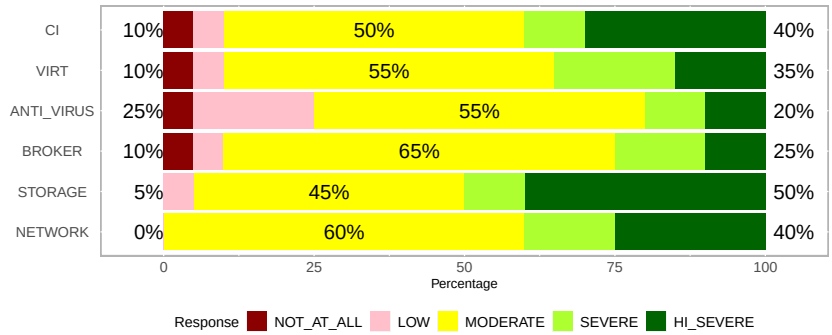

Fig. 18: Answer to RQ4: Perceptions related to the severity of task infection categories.

value is 4×10^{-07} indicating the relationship between task infection categories and the perceived frequency of the survey participants.

In the case of perceived frequency, we observe the χ^2 value to 56.7, with 10 degrees of freedom. The p-value is 2×10^{-08} indicating the relationship between task infection categories and the perceived severity of the survey participants.

> *Answer to RQ3: We observe 50% of the survey participants to find storage-related tasks to be impacted severely.*

3.4 Answer to RQ4

In this section, we answer **RQ4: What development factors correlate with task infections in Ansible scripts?** We present the results of our logistic regression models in Table 16, where we report the co-efficient esti-

Table 16: Answer to RQ4: Development Metrics and Their Correlation with Task Infection Presence

Metric	Coeff. Estimate	Error	p-value	Deviance	Odds Ratio	VIF
(Intercept)	-6.23					
Age	0.16	0.01	$< 2 \times 10^{-16}$	153.2	1.2	2.7
Bash envy	0.07	0.06	0.25	1.3	1.0	1.1
Commits	0.05	0.07	0.42	33.7	1.0	3.4
IsRole	2.75	3.56	0.44	0.57	1.5	1.0
Minor Contrib.	0.55	0.10	1.9×10^{-07}	6.0	1.1	2.1
Scatteredness	2.63	0.15	$< 2 \times 10^{-16}$	397.1	1.4	1.2

mate, standard error, $p - value$, deviance, odds ratio, and VIF. We observe a VIF of < 5 for all independent variables that show multi-colinearity does not exist between the independent variables. The McFadden R2 value for the development activity-based model is 0.05, indicating an ill-fitted model.

The metrics for which we observe correlation with the presence of task infection are age, minor contributors, and scatteredness. We observe the coefficient estimate for age, minor contributors, and scatteredness to be positive, which indicates that an increase in any of these metrics will increase the probability of task infection. The odds ratio for scatteredness is 1.4, which means increasing scatteredness by one unit raises the probability of the presence of task infection by a factor of 1.4. For age and minor contributors, the odd ratio is respectively, 1.2 and 1.1. Furthermore, the deviance value is highest for the scatteredness followed by age, and then minor contributors. Therefore, based on deviance and odds ratio, we observe scatteredness showing more correlation with task infection compared to age and minor contributors.

We further provide descriptive statistics in Table 17 for the three metrics that correlate with task infection presence. The statistic of each metric is presented as a tuple, where each tuple represents the minimum, median, average, and maximum value for each metric. The 'Infection' column provides the statistics of each metric for scripts in which at least one task infection appears. The 'Neutral' column provides the statistics of each metric for scripts in which there is no task infection. For all metrics we observe the mean to be higher for scripts with at least task infection, compared to that of scripts with no task infection. The average age, minor contributor count, and scatteredness of scripts with $>= 1$ task infections are respectively, 1.05, 1.25, and 1.19 times higher than that of scripts with no task infections.

In Table 18 we provide statistics for the source code-based model. The model excludes the following metrics because of their VIF being > 5: 'deprecated modules', 'distinct modules', 'keys', 'lines of code', 'lines of blank', 'math ops', 'parameters', 'plays', 'tokens', 'task size', and 'unique names'. The McFadden R2 value is 0.21 indicating a well-fitted model. The metrics for which $p - value < 0.01$ is highlighted in green. The source metrics that correlate with

Table 17: Descriptive Statistics of Age, Minor Contributors, and Scatteredness

Metrics	**Infection** (Min, Med., Avg., Max)	**Neutral** (Min, Med., Avg., Max)
Age	(1.0, 131.9, 331.3, 2527.0)	(1.0, 1.0, 313.8, 2486.2)
Minor contributors	(0.0, 0.0, 0.5, 11.0)	(0.0, 0.0, 0.4, 37.0)
Scatteredness	(0.0, 3.7, 3.7, 6.7)	(0.0, 3.2, 3.1, 7.4)

task infection presence are: 'average play size', 'average task size', 'blocks', 'commands', 'conditions', 'deprecated keywords', '*ensure* count', 'error handling blocks', 'error ignores', 'fact modules', 'file mode', 'includes' , '*name* with variables', 'playbook imports', 'role imports', 'role includes', 'SSH authorized keys', 'suspicious comments', 'text entropy', 'URLs', 'Var includes', and '*var* length'. The minimum, median, and maximum value for each source code metric for scripts with and without task infections is presented in Table 19.

Table 18: Answer to RQ4: Correlation of Source Code Metrics with Task Infection

Metric	**Coeff. Estimate**	**Error**	**p-value**	**Deviance**	**Odds Ratio**	**VIF**
(Intercept)	-14.1					
Average Play Size	0.18	0.03	1.9×10^{-08}	10.5	1.19	1.5
Average Task Size	0.49	0.07	3.3×10^{-11}	71.3	1.64	1.6
Blocks	0.71	0.14	1.0×10^{-06}	0.47	2.04	1.5
Commands	0.43	0.06	4.1×10^{-11}	21.5	1.54	1.7
Conditions	-0.24	0.07	0.00067	20.4	0.78	3.9
Decisions	-0.06	0.07	0.44	48.5	0.94	3.2
Deprecated Keywords	-2.76	0.73	0.0001	49.1	0.06	1.0
ensure count	-0.32	0.12	0.005	21.5	0.72	1.9
Error Handling Blocks	-1.16	0.43	0.007	20.9	0.31	1.1
Error Ignores	0.50	0.43	1.7×10^{-07}	16.8	1.65	1.3
External Modules	-0.09	0.07	0.20	21.6	0.91	1.5
Fact Modules	-0.95	0.15	4.7×10^{-10}	29.2	2.58	1.2
Files	1.57	0.68	0.02	0.06	4.81	1.0
File Mode	0.38	0.12	0.001	1.1	1.46	1.9
Filters	0.02	0.05	0.58	2.77	1.02	1.5
Includes	0.92	0.13	1.4×10^{-11}	19.5	2.50	1.1
Lines of comments	0.01	0.03	0.96	15.9	1.00	1.3
Lookups	0.06	0.10	0.56	1.4	1.06	1.1
Loops	0.08	0.06	0.20	7.3	1.08	1.5
name with variables	0.44	0.06	5.1×10^{-14}	45.2	1.56	1.2
Paths	-0.12	0.07	0.06	10.5	0.88	2.2
Playbook Imports	0.77	0.11	4.7×10^{-11}	20.7	2.15	1.1
Regexes	-0.73	0.28	0.01	4.5	0.48	1.0
Role Imports	-0.55	0.17	0.001	30.8	0.57	1.1
Role Includes	0.96	0.14	7.3×10^{-12}	28.1	2.63	1.1
Role Size	0.03	0.11	0.75	0.01	1.03	1.4
SSH Authorized Keys	2.21	0.65	0.0006	13.5	9.16	1.0
Susp. Comments	-1.29	0.24	1.1×10^{-07}	26.7	0.27	1.1
Task Imports	-0.43	0.18	0.01	9.2	0.64	1.0
Task Includes	-0.15	0.10	0.12	3.9	0.85	1.1
Text Entropy	6.43	0.43	$< 2.0 \times 10^{-16}$	590.7	21.7	2.6
URLs	1.43	0.13	$< 2.0 \times 10^{-16}$	94.0	4.18	1.1
User Interactions	-1.65	1.02	0.10	4.5	0.19	1.0
Var Includes	-1.57	0.42	0.0002	17.7	0.20	1.0
var Length	-0.91	0.05	$< 2.0 \times 10^{-16}$	186.5	0.40	1.9

In Table 20 we provide statistics for the combined model. The combined model includes development activity-based metrics and source code-based metrics. The model excludes the following metrics because of their VIF being > 5: 'average play size', 'BASH envy', 'commands', 'deprecated modules', 'distinct modules', 'keys', 'lines of code', 'lines of blank', 'math ops', 'parameters', 'plays', 'scatteredness', 'text entropy', 'tokens', 'task size', and 'unique names'. The McFadden R2 value is 0.17 indicating a well-fitted model. The metrics for which $p - value < 0.01$ is highlighted in green. According to the combined model, the metrics that correlate with the presence of task infection are: 'age', 'average task size', 'blocks', 'decisions', 'error handling blocks', 'fact modules', 'includes', 'lookups', 'loops', '*name* with variables', 'playbook imports', 'role includes', 'role size', 'SSH authorized keys', 'suspicious comments', 'task imports', 'text entropy', 'URLs', '*var* includes', and '*var* length'.

Table 19: Descriptive Statistics of Correlating Source Code Metrics

Metrics	**Infection** (Min, Med., Avg., Max)	**Neutral** (Min, Med., Avg., Max)
Average Play Size	(0.0, 0.0, 171.0)	(0.0, 0.0, 220.0)
Average Task Size	(0.0, 7.0, 221.0)	(0.0, 5.0, 177.0)
Blocks	(0.0, 0.0, 6.0)	(0.0, 0.0, 13.0)
Commands	(0.0, 0.0, 56.0)	(0.0, 0.0, 34.0)
Conditions	(0.0, 1.0, 123.0)	(0.0, 0.0, 118.0)
Deprecated Keywords	(0.0, 0.0, 1.0)	(0.0, 0.0, 21.0)
ensure count	(0.0, 0.0, 20.0)	(0.0, 0.0, 72.0)
Error Handling Blocks	(0.0, 0.0, 3.0)	(0.0, 0.0, 10.0)
Error Ignores	(0.0, 0.0, 24.0)	(0.0, 0.0, 20.0)
Fact Modules	(0.0, 0.0, 8.0)	(0.0, 0.0, 21.0)
File Mode	(0.0, 0.0, 13.0)	(0.0, 0.0, 28.0)
Includes	(0.0, 0.0, 20.0)	(0.0, 0.0, 24.0)
name with variables	(0.0, 0.0, 46.0)	(0.0, 0.0, 75.0)
Playbook Imports	(0.0, 0.0, 12.0)	(0.0, 0.0, 11.0)
Role Imports	(0.0, 0.0, 30.0)	(0.0, 0.0, 115.0)
Role Includes	(0.0, 0.0, 17.0)	(0.0, 0.0, 16.0)
SSH Authorized Keys	(0.0, 0.0, 2.0)	(0.0, 0.0, 23.0)
Susp. Comments	(0.0, 0.0, 3.0)	(0.0, 0.0, 7.0)
Text Entropy	(2.8, 5.7, 8.1)	(1.9, 5.0, 7.9)
URLs	(0.0, 0.0, 21.0)	(0.0, 0.0, 38.0)
var Includes	(0.0, 0.0, 2.0)	(0.0, 0.0, 24.0)
var Length	(0.0, 1.0, 58.0)	(0.0, 1.0, 171.0)

A summary of our logistic regression analysis is listed as follows:

- The three development activity metrics that show correlation with task infection presence are: age, minor contributors, and scatteredness.
- Amongst all selected development activity metrics scatteredness shows the most correlation.
- 21 source code metrics show correlation with task infection presence of which text entropy shows the highest correlation.
- Considering only source code metrics, with respect to correlation strength, text entropy is the most sensitive source code metric, as an unit increase

Table 20: Answer to RQ4: Correlation of Source Code and Development Activity Metrics with Task Infection

Metric	Coeff. Estimate	Error	p-value	Deviance	Odds Ratio	VIF
(Intercept)	-3.92					
Age	0.14	0.02	2.8×10^{-11}	49.2	1.14	2.8
Average Task Size	0.68	0.07	$< 2.0 \times 10^{-16}$	182.7	1.97	1.5
Blocks	0.48	0.14	0.0008	0.08	1.62	1.4
Commits	0.21	0.08	0.01	165.3	1.23	3.9
Conditions	0.13	0.06	0.04	6.9	1.14	3.5
Decisions	-0.28	0.07	0.0001	51.2	0.75	3.1
Deprecated Keywords	-1.71	0.71	0.0157	21.5	0.18	1.0
Ensure count	-0.11	0.12	0.36	1.8	0.72	2.0
Error Handling Blocks	-1.37	0.45	0.002	21.3	0.25	1.1
Error Ignores	0.85	0.09	$< 2.0 \times 10^{-16}$	53.4	2.34	1.2
External Modules	0.10	0.07	0.15	0.48	1.11	1.5
Fact Modules	1.20	0.15	7.6×10^{-15}	68.2	3.33	1.2
Files	2.13	0.68	0.02	2.2	8.43	1.0
File Mode	0.30	0.12	0.013	4.8	1.34	2.0
Filters	0.12	0.04	0.012	5.4	1.12	1.5
Includes	0.73	0.13	7.4×10^{-08}	19.5	2.07	1.1
isRole	2.14	1.51	0.15	1.7	8.51	1.0
Lines of comments	0.05	0.03	0.14	1.9	1.05	1.3
Lookups	0.28	0.10	0.008	15.3	1.32	1.1
Loops	0.19	0.06	0.004	10.8	1.20	1.5
Minor Contributors	-3.92	0.16	0.27	98.8	0.02	2.0
name with variables	0.41	0.06	8.9×10^{-12}	38.5	1.50	1.3
Paths	0.05	0.07	0.42	0.83	1.05	2.3
Playbook Imports	1.02	0.11	$< 2.0 \times 10^{-16}$	75.2	2.77	1.2
Regexes	-0.47	0.28	0.09	2.6	0.62	1.0
Role Imports	-0.25	0.16	0.12	5.9	0.78	1.2
Role Includes	0.95	0.13	2.8×10^{-12}	28.1	2.59	1.1
Role Size	0.51	0.10	5.8×10^{-07}	23.1	1.66	1.3
SSH Authorized Keys	1.85	0.64	0.004	9.9	6.35	1.0
Susp. Comments	-1.22	0.25	1.9×10^{-06}	17.1	0.29	1.1
Task Imports	-0.53	0.18	0.002	13.0	0.58	1.0
Task Includes	-0.14	0.09	0.15	3.9	0.86	1.1
URLs	1.77	0.13	$< 2.0 \times 10^{-16}$	154.7	5.90	1.1
User Interactions	-1.97	1.08	0.06	4.7	0.14	1.0
var Includes	-1.41	0.42	0.0009	17.7	0.24	1.0
var Length	-0.58	0.05	$< 2.0 \times 10^{-16}$	124.8	0.55	1.8

in text entropy will increase the probability of task infection presence by a factor of 21.7, which is higher than any other source code metric.

- Based on McFadden R2 value, the source code-based model is a better fit for task infection compared to that of the development activity-based model.
- When source code metrics are used in combination with development activity metrics, both development activity and source code metrics show correlation with task infection presence.

Answer to RQ4: We identify 3 development activity metrics and 21 source code metrics to show correlation with task infection in Ansible scripts.

Table 21: Precision, Recall and F-measure of TIDAL to Detect Task Infections for Four Security Weakness Categories

Category	Precision	Recall	F-measure
Hard-coded secret	0.93	0.98	0.96
Insecure HTTP	0.96	1.00	0.98
No Integrity Check	0.86	0.86	0.86
Unrestricted IP Address Binding	0.91	1.00	0.95
Average	0.91	0.96	0.94

3.5 Answer to RQ5

In this section, we answer **RQ5: How can we automatically task infections in Ansible scripts?** by reporting the precision, recall, and F-measure for TIDAL. We report the detection accuracy of TIDAL in Table 21. We observe the precision and recall to detect task infections is > 0.85 across all four categories. The average precision and recall is > 0.90 which gives us the confidence that TIDAL is effective in detecting task infections.

Answer to RQ5: TIDAL's precision and recall to detect task infections is > 0.85 *across all four categories.*

4 Discussion

We discuss our findings in the following subsections:

4.1 Actionability-related Implications

Our empirical study has implications related to practitioner actionability, i.e., whether or not practitioners will take action for the detected security weaknesses. In prior work, Smith et al. (2013) observed that when detecting the relevance of security weaknesses by static analysis tools, practitioners inspect if other portions of the code base use the detected weaknesses. TIDAL not only detects security weaknesses in Ansible scripts, but also identifies the tasks into which security weaknesses propagate. In this manner, TIDAL can help practitioners determine the relevance of detected security weaknesses and take necessary actions.

4.2 Implications Related to Code Inspection

Findings from Section 3.4 show age and scatteredness to be correlated with task infections. While performing code inspections, practitioners can leverage this finding to prioritize inspection efforts. For example, scripts with high age,

minor contributor count, or scatteredness can be prioritized for review, as these scripts are more likely to include task infections. For example, we observe the median age to be 131.9 times for scripts with task infections higher than that of scripts with no task infection. Practitioners can use this data as a heuristic to organize their development process of Ansible scripts.

Other source code metrics that show a correlation with task infection presence can also be used to prioritize code inspection efforts. For example, from Table 19, we observe the average task size to be 1.4 times for scripts with task infections compared to that of scripts with no task infections.

The count of affected tasks is dependent on task infections, i.e., whether or not security weaknesses propagate into tasks. If security weaknesses do not propagate into tasks, then the task is not affected. The fact that the affected tasks are smaller in count compared to the count of scripts has implications with respect to the prioritization of inspection efforts. As a smaller set of tasks are infected, with the help of `TIDAL`, the practitioner can focus first on tasks into which security weaknesses propagate.

4.3 Perceptions of Practitioners

From Figure 17, we observe >= 50% of the surveyed practitioners perceive task infection as not frequent. Such perceptions held by practitioners can potentially lead to unmitigated security weaknesses in Ansible scripts, making the provisioned computing infrastructure susceptible to security attacks. For example, if a practitioner perceives that the hard-coded secrets do not impact tasks used to set up CI servers, hard-coded secrets may remain unmitigated, allowing malicious users to conduct malicious attacks. Such attacks, unfortunately, are common: for example, hard-coded secrets were leveraged to gain unauthorized access to Uber's servers, which resulted in data exposure for 57 million customers and 600,000 Uber drivers (Miller, 2019; Schwarz, 2019). Recent research (Rahman et al., 2019; Meli et al., 2019) shows hard-coded secrets is wide-spread concern in software artifacts. To account for this concern, researchers and industry experts have advocated for the usage of secret management tools, such as Hashicorp Vault [2], by applying recommended best practices Rahman et al. (2021a) for secret management. Adoption of secret management tools can prevent exposure of hard-coded secrets in Ansible scripts. Furthermore, as organizations rely on Ansible scripts to automate their software supply chain (Ryan, 2022), unmitigated security weaknesses in Ansible scripts can lead to security attacks against an Ansible-based software supply chain.

One approach to inform practitioners about task infection is using `TIDAL`. With `TIDAL`, practitioners can detect what security weaknesses are propagating into tasks, and affecting computing infrastructure managed by these tasks.

[2] https://www.vaultproject.io/

4.4 Future Research

Our findings lay the groundwork for future research related to the secure development of Ansible scripts, which we discuss below:

- Language Agnostic Detection of Security Weaknesses: `TIDAL` is language-dependent as it uses Ansible-specific code constructs to detect security weaknesses. One possible approach could be developing an intermediate representation (IR)-based technique that can not only detect security weaknesses in Ansible scripts but also for IaC scripts developed in other programming languages, such as Chef and Puppet. As mentioned in Section 6, recent work from Saavedra and Ferreira (2023) is a right step in this direction.
- Enhanced Source Specification: Currently, `TIDAL` uses key-value pairs for the detection of security weaknesses. However, security weaknesses can propagate into tasks from other sources, e.g., from a Jinja template (Opdebeeck et al., 2022) while executing Ansible scripts. Future work can extend `TIDAL` in a manner so that it can detect security weaknesses that may originate from sources that are not key-value pairs, such as Jinja templates (Opdebeeck et al., 2022). The incorporation of practitioner feedback might also be helpful in this regard. For example, practitioners can be surveyed to understand what other sources of security weaknesses could be for Ansible tasks.
- Socio-technical Factors in Ansible-based Infrastructure Management: Findings reported in Section 3.4 show the development factors, such as age and scatteredness to correlate with the presence of task infections. We advocate for including other socio-technical factors, such as developer knowledge in software development, developer knowledge in secure coding, and contextual team factors, to investigate development factors related to task infections.
- Differences in Impacted Infrastructure for Ansible and Puppet: In Section 1, we have discussed differences with respect to impacted infrastructure between Ansible and Puppet. We posit three possible explanations: (i) the differences can be due to the studied sample projects; (ii) the differences can be due to the domain of the IaC technology in which IaC is being applied; and (iii) the differences can be due to the fact that there might be syntactic advantages for one IaC language over another, when it comes to provisioning computing infrastructure, which leads to the differences in the impacted infrastructure categories. Future research can empirically validate to what extent these explanations are substantial.
- Dynamic Analysis: While recent research has investigated static source code analysis of Ansible scripts, there is a lack of investigation on how to apply dynamic analysis of Ansible scripts. Our conjecture is that through the application of dynamic analysis, detection of security weaknesses will improve for Ansible scripts. Curating a set of Ansible playbooks that can be executed automatically is a first step towards that direction.

5 Threats to Validity

We discuss the limitations of our empirical study below:

Conclusion Validity: Our findings are limited to the task infections identified by `TIDAL`, which leverages def-use chains (Aho et al., 1986). We acknowledge that `TIDAL` may not capture all types of information flow in Ansible scripts. One type of flow that `TIDAL` is unable to detect is when secrets are provided from the command line using the `ansible` command. An example usage of this command is `ansible all -m ping -u ubuntu --ask-pass`, where a username 'ubuntu' is provided from the command line. Another type of flow is when hard-coded secrets are inherent in a value for the `shell` module. An example usage of this particular module would be `shell: > mysql -u root -NBe`. `TIDAL` accounts for one type of data flow where a detected security weakness propagates into a task. There could be other sources of security weaknesses that `TIDAL` does not account for, therefore, when applied `TIDAL`'s precision and recall may drop for those types of scripts. For example, '- u root' is a hard-coded secret, which `TIDAL` will miss, eventually reducing `TIDAL`'s recall.

Our empirical evaluation of task infection category derivation could be limited by the tasks that we mined from our set of 56 repositories. Also, our analysis is limited to `TIDAL`: application of another tool can identify task infections that we have not identified. Currently, `TIDAL` only considers data flow analysis, which is limiting and can bias the results of RQ2. `TIDAL` does not account for variable precedence with control flow analysis, which can lead to instances of false positives and false negatives. Future work can investigate if GASEL (Opdebeeck et al., 2023), which accounts for control flow analysis along with data flow analysis, can aid in better task infection detection.

Security weakness categories determined by `TIDAL` are limited to Rahman et al. (2021b)'s paper. Another limitation of our paper is not distinguishing between Ansible scripts that are used in deployment and scripts that are not deployment-related. We mitigate this limitation by using a dataset provided by prior research (Mohammad Mehedi and Rahman, 2022), which has been systematically curated.

Construct Validity: When constructing the dataset, the rater may have implicit biases that could have affected the labeling process described in Section 2.1.3. Similarly, the open coding process to derive task infection categories is also susceptible to rater bias. We mitigate both of these limitations by performing rater verification. In the case of rater verification for dataset construction and task infection category, Cohen's Kappa is, respectively, 0.78 and 0.82. Also, Reis et al. (2023) observed that practitioners do not find all security weakness categories, such as hard-coded user names to be relevant. As such, `TIDAL`'s precision may reduce when applied to a dataset, which is based on practitioner perceptions, even if the security weakness categories are valid.

Our use of names for tasks in the open coding process is limiting, as IaC scripts, such as Ansible scripts, are susceptible to linguistic inconsistencies.

Also, our use of task content when the name is not available can limit the category derivation process.
External Validity: Our empirical study is susceptible to external validity as our analysis is limited to datasets collected from OSS repositories. `TIDAL` can generate false positives and false negatives for datasets not used in the paper, which in turn can influence results presented in Sections 3.5, 3.2, 3.3, and 3.4. The dataset labeling process used for RQ1 is susceptible to generating false negatives that can influence the results of our empirical study. We mitigate this limitation by using another rater who inspected a set of 50 Ansible to identify false negatives. The rater did not identify any false negatives. Furthermore, as `TIDAL` does not consider variable presence there is the possibility of security weaknesses being replaced with values that are overwritten, which in turn converting the detected security weaknesses as false positives. These false positive instances will be irrelevant even if the variable propagates into a task. In such cases, `TIDAL` will generate false positives. Our paper is susceptible to external validity with respect to survey analysis as we have analyzed survey responses from 23 practitioners. A larger survey respondent population could have improved the generalizability of our survey-related findings.

6 Related Work

Our paper is related to prior research investigating code elements pertinent to the quality assurance for Ansible scripts. Dalla Palma et al. (2020) proposed a suite of 46 metrics that include code properties of Ansible scripts that can be used to identify defective Ansible scripts. In another paper, unlike for projects that use general-purpose programming languages (Rahman and Devanbu, 2013), Dalla Palma et al. (2022) found code metrics to outperform development activity metrics for predicting defects in Ansible scripts. Opdebeeck et al. (2022) identified code smells that can cause defects in Ansible scripts. Hassan and Rahman (2022) derived the taxonomy of defects observed in Ansible test scripts. Kokuryo et al. (2020) identified execution of external scripts as a quality concern in Ansible scripts. Specific categories of defects, such as security defects, have also garnered interest. Rahman et al. (2021b) derived a taxonomy of security weaknesses in Ansible scripts and constructed a tool called SLAC to detect security weaknesses in Ansible scripts automatically. Rahman et al. (2021b)'s paper was replicated by Hortlund (2021), who reported the security weakness density to be less than that reported by Rahman et al. (Rahman et al., 2021b), due to false positives generated by SLAC. Rahman et al. (2021a) in another paper further built upon their prior work (Rahman et al., 2021b) to identify best practices to remove security weaknesses in Ansible scripts, such as hard-coded secrets. Findings from Rahman et al. (Rahman et al., 2021b)'s paper was integrated into course curriculum by educators (2022), who observed a learning approach called authentic learning (Lombardi and Oblinger, 2007) to be useful for educating students about security weaknesses in Ansible scripts. Saavedra and Ferreira (2023) con-

structed an intermediate representation to improve security weakness detection for Ansible scripts. Reis et al. (2023) incorporated practitioner feedback into an existing security linter for Puppet and found such incorporation to improve security weakness detection for Puppet scripts. Opdebeeck et al. (2023) argued the need for incorporating control and data flow analysis for security weakness detection in Ansible scripts. Hu et al. (2023) characterized static analysis alerts for Terraform manifests. Our paper is different from these publications in the following aspects: (i) we study the phenomenon of task infection in Ansible scripts that have not been investigated before; (ii) we categorize and quantify task infections in Ansible scripts that have not been investigated before; and (iii) we quantify empirical evidence that demonstrates the relationship between task infection presence and development activity metrics as well as task infection presence and source code metrics.

From the aforementioned discussion, we observe a lack of research that has systematically investigated task infection in Ansible scripts, which we address in our paper.

7 Conclusion

As practitioners use Ansible scripts for managing computing infrastructure at scale, unmitigated security weaknesses in Ansible scripts can allow malicious users to conduct security attacks. We hypothesize that we can identify security weaknesses accurately by detecting task infection, i.e., propagation of security weaknesses into tasks. We construct `TIDAL` to detect and characterize task infections in Ansible scripts. With `TIDAL` we identify 1,805 task infections in 27,213 scripts. We identify six categories of task infections, amongst which tasks used to manage data storage infrastructure are the most frequent. From our survey with 23 practitioners, we observe tasks used to manage data storage to be perceived as the most severe. Also, we observe age and scatteredness to correlate with task infections.

Based on our findings, we recommend the detection of task infections for security static analysis of Ansible scripts because it provides information on how detected security weaknesses are impacting the computing infrastructure that is managed with Ansible. We hope our paper will help future research in the area of secure Ansible script development.

Disclaimers

Conflict of Interests/Competing Interests The authors have no relevant financial or non-financial interests to disclose.
Data Availability Statements Dataset and source code used in our paper is publicly available online (Rahman, 2023).

Acknowledgements We thank the PASER group at Auburn University for their valuable feedback. This research was partially funded by the U.S. National Science Foundation (NSF)

Award # 2247141, Award # 2310179, Award # 2312321, and the U.S. National Security Agency (NSA) Award # H98230-21-1-0175.

References

Agrawal A, Rahman A, Krishna R, Sobran A, Menzies T (2018) We don't need another hero?: The impact of "heroes" on software development. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ACM, New York, NY, USA, ICSE-SEIP '18, pp 245–253, DOI 10.1145/3183519.3183549, URL http://doi.acm.org/10.1145/3183519.3183549

Aho AV, Sethi R, Ullman JD (1986) Compilers, principles, techniques. Addison wesley 7(8):9

Akond R, Laurie W (2019) Source code properties of defective infrastructure as code scripts. Information and Software Technology DOI https://doi.org/10.1016/j.infsof.2019.04.013

Ansible (2020) Ansible Documentation. https://docs.ansible.com/, [Online; accessed 19-December-2020]

Ansible (2022) Ansible best practices. https://docs.ansible.com/ansible/2.8/, [Online; accessed 10-Sep-2022]

Banavar G, Chandra TD, Strom RE, Sturman DC (1999) A case for message oriented middleware. In: Proceedings of the 13th International Symposium on Distributed Computing, Springer-Verlag, Berlin, Heidelberg, p 1–18

Bird C, Nagappan N, Murphy B, Gall H, Devanbu P (2011) Don't touch my code! examining the effects of ownership on software quality. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE '11, p 4–14, DOI 10.1145/2025113.2025119, URL https://doi.org/10.1145/2025113.2025119

Borovits N, Kumara I, Di Nucci D, Krishnan P, Palma SD, Palomba F, Tamburri DA, Heuvel WJvd (2022) Findici: Using machine learning to detect linguistic inconsistencies between code and natural language descriptions in infrastructure-as-code. Empirical Software Engineering 27(7):178

Carver JC (2010) Towards reporting guidelines for experimental replications: A proposal. In: 1st international workshop on replication in empirical software engineering, vol 1, pp 1–4

Cohen J (1960) A coefficient of agreement for nominal scales. Educational and Psychological Measurement 20(1):37–46, DOI 10.1177/001316446002000104, URL http://dx.doi.org/10.1177/001316446002000104, http://dx.doi.org/10.1177/001316446002000104

Cohen P, West SG, Aiken LS (2014) Applied multiple regression/correlation analysis for the behavioral sciences. Psychology press

Cozens B (2022) 10 habits of great ansible users. https://www.redhat.com/sysadmin/10-great-ansible-practices, [Online; accessed 10-Sep-2022]

Cramer D, Howitt DL (2004) The Sage dictionary of statistics: a practical resource for students in the social sciences. Sage

Da Silva FQ, Suassuna M, França ACC, Grubb AM, Gouveia TB, Monteiro CV, dos Santos IE (2014) Replication of empirical studies in software engineering research: a systematic mapping study. Empirical Software Engineering 19:501–557

Dalla Palma S, Di Nucci D, Palomba F, Tamburri DA (2020) Toward a catalog of software quality metrics for infrastructure code. Journal of Systems and Software 170:110726

Dalla Palma S, Di Nucci D, Palomba F, Tamburri DA (2022) Within-project defect prediction of infrastructure-as-code using product and process metrics. IEEE Transactions on Software Engineering 48(6):2086–2104, DOI 10.1109/TSE.2021.3051492

Davis V (2019) Ansible role patterns and anti-patterns by lee garrett, its debian maintainer. https://hub.packtpub.com/ansible-role-patterns-and-anti-patterns-by-lee-garrett-its-debian-maintainer/, [Online; accessed 11-Sep-2022]

Droms R (1999) Automated configuration of tcp/ip with dhcp. IEEE Internet Computing 3(4):45–53, DOI 10.1109/4236.780960

Duvall P, Matyas SM, Glover A (2007) Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series). Addison-Wesley Professional

Easterbrook S, Singer J, Storey MA, Damian D (2008) Selecting Empirical Methods for Software Engineering Research, Springer London, London, pp 285–311

Gelman A, Hill J (2006) Data analysis using regression and multilevel/hierarchical models. Cambridge university press

Greenwood PE, Nikulin MS (1996) A guide to chi-squared testing, vol 280. John Wiley & Sons

Hassan AE (2009) Predicting faults using the complexity of code changes. In: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, ICSE '09, pp 78–88, DOI 10.1109/ICSE.2009.5070510, URL http://dx.doi.org/10.1109/ICSE.2009.5070510

Hortlund A (2021) Security smells in open-source infrastructure as code scripts: A replication study

Hosmer Jr DW, Lemeshow S, Sturdivant RX (2013) Applied logistic regression, vol 398. John Wiley & Sons

Hu H, Bu Y, Wong K, Sood G, Smiley K, Rahman A (2023) Characterizing static analysis alerts for terraform manifests: An experience report. In: 2023 IEEE Secure Development Conference (SecDev), IEEE Computer Society, Los Alamitos, CA, USA, pp 7–13, DOI 10.1109/SecDev56634.2023.00014, URL https://doi.ieeecomputersociety.org/10.1109/SecDev56634.2023.00014

Humble J, Farley D (2010) Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation, 1st edn. Addison-Wesley Professional

Jenkins (2022) Jenkins. https://www.jenkins.io/, [Online; accessed 23-Jan-2022]

Kitchenham BA, Pfleeger SL (2008) Personal Opinion Surveys, Springer London, London, pp 63–92. DOI 10.1007/978-1-84800-044-5_3, URL https://doi.org/10.1007/978-1-84800-044-5_3

Kokuryo S, Kondo M, Mizuno O (2020) An empirical study of utilization of imperative modules in ansible. In: 2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS), pp 442–449, DOI 10.1109/QRS51102.2020.00063

Krein JL, Knutson CD (2010) A case for replication : Synthesizing research methodologies in software engineering

Krishna R, Agrawal A, Rahman A, Sobran A, Menzies T (2018) What is the connection between issues, bugs, and enhancements?: Lessons learned from 800+ software projects. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ACM, New York, NY, USA, ICSE-SEIP '18, pp 306–315, DOI 10.1145/3183519.3183548, URL http://doi.acm.org/10.1145/3183519.3183548

Labs P (2021) Puppet Documentation. https://docs.puppet.com/, [Online; accessed 01-July-2021]

Landis JR, Koch GG (1977) The measurement of observer agreement for categorical data. Biometrics 33(1):159–174, URL http://www.jstor.org/stable/2529310

Lombardi MM, Oblinger DG (2007) Authentic learning for the 21st century: An overview. Educause learning initiative 1(2007):1–12

Long JS, Freese J (2006) Regression models for categorical dependent variables using Stata, vol 7. Stata press

Mann HB, Whitney DR (1947) On a test of whether one of two random variables is stochastically larger than the other. The Annals of Mathematical Statistics 18(1):50–60, URL http://www.jstor.org/stable/2236101

Meli M, McNiece MR, Reaves B (2019) How bad can it git? characterizing secret leakage in public github repositories. In: NDSS

Menard S (2002) Applied logistic regression analysis. 106, Sage

Miller M (2019) Hardcoded and Embedded Credentials are an IT Security Hazard – Here's What You Need to Know. https://www.beyondtrust.com/blog/entry/hardcoded-and-embedded-credentials-are-an-it-security-hazard-heres-what-you-need-to-know, [Online; accessed 17-Jan-2022]

Mohammad Mehedi H, Rahman A (2022) As code testing: Characterizing test quality in open source ansible development. In: 2022 15th IEEE Conference on Software Testing, Verification and Validation (ICST), IEEE Computer Society, Los Alamitos, CA, USA, URL https://akondrahman.github.io/publication/icst2022

Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density. In: Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005., pp 284–292, DOI 10.1109/ICSE.2005.1553571

Opdebeeck R, Zerouali A, De Roover C (2022) Smelly variables in ansible infrastructure code: Detection, prevalence, and lifetime. In: 2022 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), IEEE

Opdebeeck R, Zerouali A, Roover CD (2023) Control and data flow in security smell detection for infrastructure as code: Is it worth the effort? In: 2023 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)

redhat performance (2022) redhat-performance/satperf. https://github.com/redhat-performance/satperf, [Online; accessed 02-July-2022]

Radjenović D, Heričko M, Torkar R, Živkovič A (2013) Software fault prediction metrics: A systematic literature review. Information and software technology 55(8):1397–1418

Rahman A (2023) Verifiability package for paper. https://figshare.com/s/c9d7b8aa973f53f02234, [Online; accessed 25-August-2023]

Rahman A, Parnin C (2023) Detecting and characterizing propagation of security weaknesses in puppet-based infrastructure management. IEEE Transactions on Software Engineering 49(06):3536–3553, DOI 10.1109/TSE.2023.3265962

Rahman A, Williams L (2021) Different kind of smells: Security smells in infrastructure as code scripts. IEEE Security Privacy 19(3):33–41, DOI 10.1109/MSEC.2021.3065190

Rahman A, Agrawal A, Krishna R, Sobran A (2018a) Characterizing the influence of continuous integration: Empirical results from 250+ open source and proprietary projects. In: Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics, ACM, New York, NY, USA, SWAN 2018, pp 8–14, DOI 10.1145/3278142.3278149, URL http://doi.acm.org/10.1145/3278142.3278149

Rahman A, Mahdavi-Hezaveh R, Williams L (2018b) A systematic mapping study of infrastructure as code research. Information and Software Technology DOI https://doi.org/10.1016/j.infsof.2018.12.004, URL http://www.sciencedirect.com/science/article/pii/S0950584918302507

Rahman A, Parnin C, Williams L (2019) The seven sins: security smells in infrastructure as code scripts. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, pp 164–175

Rahman A, Farhana E, Parnin C, Williams L (2020a) Gang of eight: A defect taxonomy for infrastructure as code scripts. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '20, p 752–764, DOI 10.1145/3377811.3380409, URL

https://doi.org/10.1145/3377811.3380409
Rahman A, Farhana E, Williams L (2020b) The 'as code'activities: development anti-patterns for infrastructure as code. Empirical Software Engineering 25(5):3430–3467
Rahman A, Barsha FL, Morrison P (2021a) Shhh!: 12 practices for secret management in infrastructure as code. In: 2021 IEEE Secure Development Conference (SecDev), pp 56–62, DOI 10.1109/SecDev51306.2021.00024
Rahman A, Rahman MR, Parnin C, Williams L (2021b) Security smells in ansible and chef scripts: A replication study. ACM Trans Softw Eng Methodol 30(1), DOI 10.1145/3408897, URL https://doi.org/10.1145/3408897
Rahman A, Shamim SI, Shahriar H, Wu F (2022) Can We use Authentic Learning to Educate Students About Secure Infrastructure as Code Development?, Association for Computing Machinery, New York, NY, USA. URL https://akondrahman.github.io/publication/iticse2022
Rahman F, Devanbu P (2013) How, and why, process metrics are better. In: 2013 35th International Conference on Software Engineering (ICSE), pp 432–441, DOI 10.1109/ICSE.2013.6606589
RedHat (2022a) Customer Case Study - NEC. https://www.ansible.com/hubfs/pdf/Ansible-Case-Study-NEC.pdf, [Online; accessed 12-Sep-2022]
RedHat (2022b) Customer Case Study - NetApp. https://www.ansible.com/hubfs/2018_Content/RH-netapp-case-study.pdf, [Online; accessed 02-Oct-2022]
Reis S, Abreu R, d'Amorim M, Fortunato D (2023) Leveraging practitioners' feedback to improve a security linter. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, Association for Computing Machinery, New York, NY, USA, ASE '22, DOI 10.1145/3551349.3560419, URL https://doi.org/10.1145/3551349.3560419
Ryan J (2022) Ansible automation platform: Private automation hub. https://people.redhat.com/bdumont/Central-Region-Lunch-n-Learns/Ansible_Automation_Platform_Private_Automation_Hub.pdf, [Online; accessed 10-Dec-2022]
Saavedra N, Ferreira JaF (2023) Glitch: Automated polyglot security smell detection in infrastructure as code. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, Association for Computing Machinery, New York, NY, USA, ASE '22, DOI 10.1145/3551349.3556945, URL https://doi.org/10.1145/3551349.3556945
Saldaña J (2015) The coding manual for qualitative researchers. Sage
Schwarz J (2019) Hardcoded and Embedded Credentials are an IT Security Hazard – Here's What You Need to Know. https://www.beyondtrust.com/blog/entry/hardcoded-and-embedded-credentials-are-an-it-security-hazard-heres-what-you-need-to-know, [Online; accessed 02-July-2021]
Shull FJ, Carver JC, Vegas S, Juristo N (2008) The role of replications in empirical software engineering. Empirical software engineering 13:211–218

Smith E, Loftin R, Murphy-Hill E, Bird C, Zimmermann T (2013) Improving developer participation rates in surveys. In: 2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), pp 89–92, DOI 10.1109/CHASE.2013.6614738

Smith J, Johnson B, Murphy-Hill E, Chu B, Lipford HR (2015) Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2015, p 248–259, DOI 10.1145/2786805.2786812, URL https://doi.org/10.1145/2786805.2786812

Tan PN, Steinbach M, Kumar V (2005) Introduction to Data Mining, (First Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

Yevgeniy Brikman (2016) Why we use terraform and not chef, puppet, ansible, saltstack, or cloudformation. https://blog.gruntwork.io/why-we-use-terraform-and-not-chef-puppet-ansible-saltstack-or-cloudformation-7989dad2865c, [Online; accessed 24-July-2023]