Security Smells in Ansible and Chef Scripts: A Replication Study

AKOND RAHMAN, Tennessee Technological University, USA
MD RAYHANUR RAHMAN, CHRIS PARNIN, and LAURIE WILLIAMS,
NC State University, USA

Context: Security smells are recurring coding patterns that are indicative of security weakness and require further inspection. As infrastructure as code (IaC) scripts, such as Ansible and Chef scripts, are used to provision cloud-based servers and systems at scale, security smells in IaC scripts could be used to enable malicious users to exploit vulnerabilities in the provisioned systems. **Goal:** *The goal of this article is to help practitioners avoid insecure coding practices while developing infrastructure as code scripts through an empirical study of security smells in Ansible and Chef scripts.* **Methodology:** We conduct a replication study where we apply qualitative analysis with 1,956 IaC scripts to identify security smells for IaC scripts written in two languages: Ansible and Chef. We construct a static analysis tool called Security Linter for Ansible and Chef scripts (SLAC) to automatically identify security smells in 50,323 scripts collected from 813 open source software repositories. We also submit bug reports for 1,000 randomly selected smell occurrences. **Results:** We identify two security smells not reported in prior work: missing default in case statement and no integrity check. By applying SLAC we identify 46,600 occurrences of security smells that include 7,849 hard-coded passwords. We observe agreement for 65 of the responded 94 bug reports, which suggests the relevance of security smells for Ansible and Chef scripts amongst practitioners. **Conclusion:** We observe security smells to be prevalent in Ansible and Chef scripts, similarly to that of the Puppet scripts. We recommend practitioners to rigorously inspect the presence of the identified security smells in Ansible and Chef scripts using (i) code review, and (ii) static analysis tools.

CCS Concepts: • **Security and privacy → Software security engineering**;

Additional Key Words and Phrases: Ansible, chef, configuration as code, configuration scripts, devops, devsecops, empirical study, infrastructure as code, insecure coding, security, smell, static analysis

1 INTRODUCTION

Infrastructure as code (IaC) is the practice of using automated scripting to provision and configure their development environment and servers at scale [16]. Similar to software source code, recommended software engineering practices, such as version control and testing, are expected to be applied to implement the practice of IaC. IaC tool vendors, such as Ansible[1] and Chef,[2] provide programming utilities to implement the practice of IaC. The use of IaC scripts has resulted in benefits for information technology (IT) organizations. For example, the use of IaC scripts helped the National Aeronautics and Space Administration to reduce its multi-day patching process to 45 minutes [3]. Using IaC scripts, application deployment time for the Borsa Istanbul, Turkey, stock exchange reduced from ~10 days to an hour [23]. With IaC scripts, Ambit Energy increased their deployment frequency by a factor of 1,200 [32]. The Enterprise Strategy Group surveyed practitioners and reported the use of IaC scripts to help IT organizations gain 210% in time savings and 97% in cost savings on average [25].

Despite reported benefits, IaC scripts can be susceptible to security weakness. In our recent work, we identified security smells for Puppet scripts [37]. Security smells are recurring coding patterns that are indicative of security weakness and require further inspection [37]. We identified 21,201 occurrences of seven security smells that include 1,326 occurrences of hard-coded passwords in 15,232 Puppet scripts. Our prior research showed relevance of the identified security smells amongst practitioners as well: From 212 responses we observe practitioners to agree with 148 occurrences.

IT organizations may use other languages, such as Ansible, Chef, and Terraform,[3] for which our previous categorization of security smells reported in prior work [37] may not hold. A replication of our prior work for other languages, such as Ansible and Chef, may have value for practitioners as well as for research as we study the generalizability and robustness of IaC security smells in a larger variety of contexts. A 2019 survey with 786 practitioners reported Ansible as the most popular language to implement IaC, followed by Chef.[4,5] As usage of Ansible and Chef is getting increasingly popular amongst practitioners, identification of security smells could have relevance to practitioners in mitigating insecure coding practices in IaC.

Our prior research [37] is not exhaustive and may not capture security smells that exist for other languages. Let us consider Figure 1 in this regard. In Figure 1, we present an actual Ansible code snippet downloaded from an open source software (OSS) repository.[6] In the code snippet, we observe the "gpgcheck" parameter is assigned "no," indicating that while downloading the "nginx" package, the "yum" package manager will not check the contents of the downloaded package.[7] Not checking the content of a downloaded package is related to a security weakness called "Download of Code Without Integrity Check (CWE-494)."[8] According to Common Weakness Enumeration (CWE), not specifying integrity check may help malicious users to "*execute attacker-controlled commands, read or modify sensitive resources, or prevent the software from functioning correctly for legitimate users.*"

Existence and persistence of security smells similar to Figure 1 in IaC scripts provide attackers opportunities to attack the provisioned system. We hypothesize through a replication [45] of our

---

[1]https://www.ansible.com/.
[2]https://www.chef.io/chef/.
[3]https://www.terraform.io/.
[4]https://info.flexerasoftware.com/SLO-WP-State-of-the-Cloud-2019.
[5]https://www.techrepublic.com/article/ansible-overtakes-chef-and-puppet-as-the-top-cloud-configuration-management-tool/.
[6]https://git.openstack.org/cgit/openstack/openstack-ansible-ops/.
[7]https://docs.ansible.com/ansible/2.3/yum_repository_module.html.
[8]https://cwe.mitre.org/data/definitions/494.html.

```
- name: Add nginx repo to yum sources list
  yum_repository:
    name: "nginx"
    file: "nginx"
    description: "NGINX repo"
    baseurl: "{{ elastic_nginx_repo.repo }}"
    state: "{{ elastic_nginx_repo.state }}"
    enabled: yes
    gpgcheck: no
```

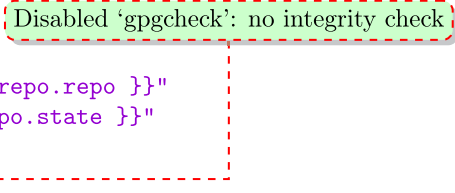Disabled 'gpgcheck': no integrity check

Fig. 1. An example Ansible script where integrity check is not specified.

prior work, we can systematically identify security smells for other languages namely, Ansible and Chef.

The goal of this article is to help practitioners avoid insecure coding practices while developing infrastructure as code scripts through an empirical study of security smells in Ansible and Chef scripts. We answer the following research questions:

- **RQ1:** What security smells occur in Ansible and Chef scripts?
- **RQ2:** How frequently do security smells occur for Ansible and Chef scripts?
- **RQ3:** How do practitioners perceive the identified security smell occurrences for Ansible and Chef scripts?

We build on prior research [37] related to security smells for IaC scripts on Puppet, and investigate what security smells for two languages used to implement the practice of IaC, namely Ansible and Chef. We conduct a differentiated replication [19, 21] of our prior work [37], where we use an experimental setup that differs from our prior work using Ansible and Chef scripts. We apply qualitative analysis [54] on 1,101 Ansible scripts and 855 Chef scripts to determine security smells. Next, we construct a static analysis tool called Security Linter for Ansible and Chef scripts (SLAC) [37] to automatically identify the occurrence of these security smells in 14,253 Ansible and 36,070 Chef scripts collected by respectively mining 365 and 448 OSS repositories. We calculate smell density for each type of security smell in the collected IaC scripts. We submit bug reports for 1,000 randomly selected smell occurrences for Ansible and Chef to assess the relevance of the identified security smells.

Contributions: Compared to our prior research [37] in which we reported findings specific to Puppet, we make the following additional contributions:

- A list of security smells for Ansible and Chef scripts that include two categories not reported in prior work [37];
- An evaluation of security smell frequency occuring in Ansible and Chef scripts. As a result of this evaluation, we have created a benchmark of how frequently security smells appear for Ansible and Chef, which was missing for the two languages. The frequency of identified security smells for Ansible and Chef scripts can be used as a measuring stick by practitioners and researchers alike;
- A detailed discussion on how practitioner responses from bug reports can drive actionable detection and repair of Ansible and Chef security smells. In our prior work, we did not discuss how the practitioner's responses in bug reports can guide tools for actionable detection and repair;
- An empirically validated tool (SLAC) that automatically detects occurrences of security smells for Ansible and Chef scripts. The tool that we constructed as part of prior work will
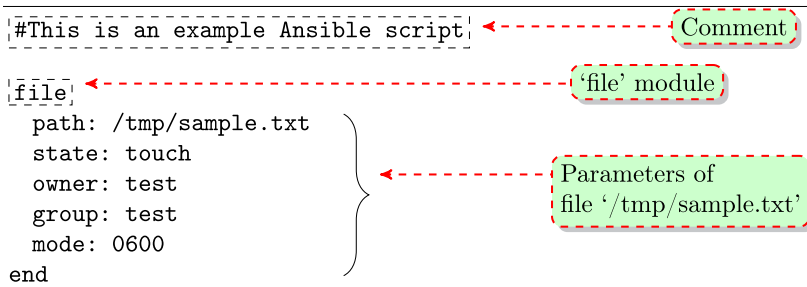
```
#This is an example Ansible script    ◄----------  Comment

file  ◄---------------------------------  'file' module
  path: /tmp/sample.txt
  state: touch
  owner: test                       ◄---------  Parameters of
  group: test                                   file '/tmp/sample.txt'
  mode: 0600
end
```

Fig. 2. Annotation of an example Ansible script.

not work for Ansible and Chef scripts. The "Parser" component of SLAC differs from that of
"SLIC" that we built in our prior work. The "Rule Engine" component of SLAC differs from
that of SLIC [37], as, unlike Puppet, which uses attributes, Ansible and Chef respectively
use "Keys" and "Properties"; and
- A discussion of differences among the three IaC languages: Ansible, Chef, and Puppet. In
our prior work, we provided background on Puppet scripts only and did not discuss the
differences among Ansible, Chef, and Puppet.

We organize the rest of the article as follows: We provide background information with related
work discussion in Section 2. We describe the methodology and the definitions of identified secu-
rity smells in Section 3. We describe the methodology to construct and evaluate SLAC in Section 4.
In Section 5, we describe the methodology for our empirical study. We report our findings in Sec-
tion 6, followed by a discussion in Section 7. We describe limitations in Section 8 and conclude our
article in Section 9.

2 BACKGROUND AND RELATED WORK

We provide background information with related work discussion in this section.

2.1 Background

In this section, we provide background on Ansible and Chef scripts, along with CWE, as we use
CWE to validate our qualitative process described in Section 3.1.

2.1.1 Ansible and Chef Scripts. We provide a brief background on Ansible and Chef scripts,
which is relevant to conduct our empirical study. Both Ansible and Chef provide multiple libraries
to manage infrastructure and system configurations. In the case of Ansible, developers can manage
configurations using "playbooks," which uses YAML files to manage configurations. For example,
as shown in Figure 2, an empty file "/tmp/sample.txt" is created using the "file" module provided
by Ansible. The properties of the file such as path, owner, and group can also be specified. The
"state" property provides options to create an empty file using the "touch" value.

In the case of Chef, configurations are specified using "recipes," which are domain-specific
Ruby scripts. Dedicated libraries are also available to maintain certain configurations. As shown in
Figure 3, using the "file" resource, an empty file "/var/sample.txt" is created. The "content" property
is used to specify the content of the file is empty.

2.1.2 Differences among Ansible, Chef, and Puppet. The three languages differ from each other
with respect to execution order, perceived codebase maintenance, requiring additional agent soft-
ware installation, style, and syntax. We discuss each of these differences below and also present a
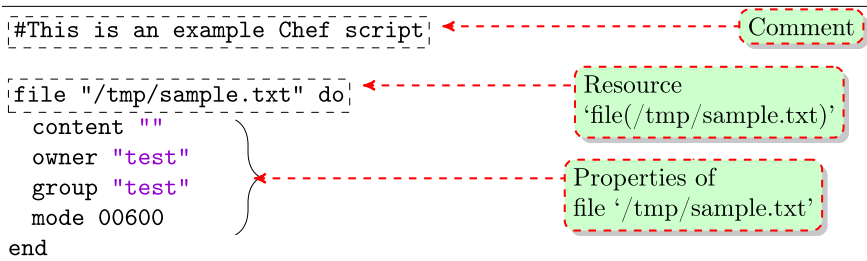summary of the differences in Table 1.

```
#This is an example Chef script          ◀----------------  Comment

file "/tmp/sample.txt" do          ◀----------  Resource
  content ""                                    'file(/tmp/sample.txt)'
  owner "test"
  group "test"          ◀----------  Properties of
  mode 00600                                    file '/tmp/sample.txt'
end
```

Fig. 3. Annotation of an example Chef script.

Table 1. Summary of Differences among Ansible, Chef, and Puppet Scripts

Lang.	Execution Order	Perceived Maint.	Add. agent	Style	Syntax	Construction
Ansible	Provisioning dependent on ordering of code	High	No	Declarative	YAML	Python
Chef	Provisioning dependent on ordering of code	High	Yes	Declarative	Ruby	Ruby
Puppet	Provisioning independent on ordering of code	Low	Yes	Procedural	Puppet DSL	Ruby

- *Construction*: Ansible is created with Python, whereas Chef and Puppet are created using Ruby.
- *Execution order*: For procedural configuration languages, such as Ansible and Chef, understanding of the order in which tasks are executed is important, because specifying a different order might provision the desired infrastructure differently. However, for Puppet, the current code state provides a clear view of what will be the configurations of the provisioned infrastructure.
- *Perceived codebase maintenance*: Practitioners [55] perceive Ansible and Chef code bases to be large and incur more maintenance overhead, as previously written Ansible and Chef code might be obsolete after a certain period of time. 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. Unlike Ansible and Chef, Puppet code is a direct reflection of the current state of the provisioned infrastructure, and practitioner might not need to write new code to be consistent with the current state of the provisioned infrastructure.
- *Requiring additional agent software*: For Chef and Puppet users installation of additional agent software is required for each of the servers that the practitioner wants to configure [4, 9]. Typically, the agents run as background services and executes necessary updates to the provision infrastructure when needed. Practitioners [55] perceive use of agent software to have limitations related to maintenance. For example, if a defect occurs, then the practitioner needs to troubleshoot the scripts, the installed agents, as well as the communication amongst the installed agents. For Ansible, installation of additional agent software is not required.
- *Style*: Ansible and Chef scripts are developed using a procedural style. Practitioners write Ansible and Chef scripts in a step-by-step manner so that the desired end state is reached. Unlike Ansible and Chef, Puppet is developed using a declarative style, where the desired state is specified first, and the Puppet tool itself is responsible to reach the desired state. In both cases, the desired state refers to the state of the provisioned computing infrastructure. For example, in the case of Figures 2 and 3, the desired state is to create an empty text file.

- *Syntax*: Ansible, Chef, and Puppet respectively use YAML, Ruby, and Puppet domain specific language (DSL) as their syntax. The differences in the syntax of the programming languages determine the expressiveness for the programming languages. For example, practitioners [55] have reported that declarative languages, such as Ansible and Chef, can be limiting to conduct certain DevOps-related tasks, such as gradual rollouts and zero-downtime deployment.

2.1.3 Common Weakness Enumeration. CWE is a community-driven database for software security weaknesses and vulnerabilities [27]. The goal of creating this database is to understand security weaknesses in software, create automated tools so that security weaknesses in software can be automatically identified and repaired, and create a common baseline standard for security weakness identification, mitigation, and prevention efforts [27]. The database is owned by the MITRE Corporation, with support from US-CERT and the National CyberSecurity Division of the United States Department of Homeland Security [27].

2.1.4 Differentiated Replication in Software Engineering. We conduct a differentiated replication [21] of our prior work [37]. Krein and Knutson [21] constructed a replication taxonomy for software engineering research. Their taxonomy included four categories of replication, namely, strict replication, differentiated replication, dependent replication, and independent replication. In strict replication, protocols of a prior research study is strictly followed as possible. In differentiated replication, the protocol of the prior research study is intentionally altered by the researchers. Dependent replication refers to research studies that is designed with reference to one or more prior research studies. Independent replication answers the same research questions as a prior research study, but it conducted without knowledge of, or deference to the prior research study. Our research article focuses on Ansible and Chef scripts, which necessitates alteration in the study design of our prior research article on security smells [37].

2.2 Related Work

For IaC scripts, we observe a lack of studies that investigate coding practices with security consequences. For example, Sharma et al. [48], Schwarz [46], and Bent et al. [52], in separate studies investigated code maintainability aspects of Chef and Puppet scripts. Jiang and Adams [18] investigated the co-evolution of IaC scripts and other software artifacts, such as build files and source code. Rahman and Williams [40] characterized defective IaC scripts using text mining and created prediction models using text feature metrics. Rahman et al. [38] surveyed practitioners to investigate which factors influence usage of IaC tools. Rahman et al. [36] conducted a systematic mapping study with 32 IaC-related publications and observed lack in security-related research in the domain of IaC. Rahman and Williams [41] identified 10 code properties in IaC scripts that show correlation with defective IaC scripts. Hanappi et al. [15] investigated how convergence of IaC scripts can be automatically tested and proposed an automated model-based test framework. Rahman et al. [34] also constructed a defect taxonomy for IaC scripts that included eight defect categories. In another work Rahman et al. [35] identified five development anti-patterns for IaC scripts. In this article, we build upon the research conducted by Rahman et al. [37]'s research, which identified seven types of security smells that are indicative of security weaknesses in IaC scripts. They identified 21,201 occurrences of security smells that include 1,326 occurrences of hard-coded passwords. The three languages differ from each other with respect to execution order, perceived codebase maintenance, requiring additional agent software installation, style, and syntax. Differences in IaC languages along with the need to advance the science of IaC script quality motivate us to conduct our research. We replicate Rahman et al. [37]'s research for Ansible and Chef scripts.

Table 2. Summary Statistics of the Collected Repositories Used in RQ1

Lang.	Duration	Repo. Cnt	Dev. Cnt	Com. Cnt	IaC Cnt.	IaC Size
Ansible	02-2014 to 2018-11	16	1,175	20,294	1,101	138,679
Chef	05-2011 to 2018-11	11	650	4,758	855	124,808

3 SECURITY SMELLS

A code smell is a recurrent coding pattern that is indicative of potential maintenance problems [14]. A code smell may not always have bad consequences but still deserves attention, as a code smell may be an indicator of a problem [14]. Our article focuses on identifying security smells. Security smells are recurring coding patterns that are indicative of security weakness and require further inspection [37].

We conduct a differentiated replication of our prior research, where we alter the research questions and methodology for Puppet scripts and apply the methodology for Ansible and Chef scripts. We exclude the analysis of lifetime, because before quantifying the lifetime of security smells, we wanted to understand (i) what categories of security smells exist, (ii) if security smells are frequent, and (iii) if the identified security smells have relevance to practitioners. Without establishing the groundwork that addresses all these factors, lifetime analysis would not have been relevant for practitioners.

We describe the methodology to derive security smells in IaC scripts, followed by the definitions and examples for the identified security smells.

3.1 RQ1: What security smells occur in Ansible and Chef scripts?

Data Collection: We collect a set of Ansible and Chef scripts to determine security smells for each language. We collect 1,101 Ansible scripts that we use to determine the security smells from 16 OSS repositories maintained by Openstack. We collect 855 Chef scripts from 10 repositories maintained by Openstack. We select Openstack as Openstack provides utilities related to cloud computing and have made their source code online. Our assumption is that by collecting Ansible and Chef scripts from the repositories we will be able to obtain a sufficient amount of Ansible and Chef scripts to perform qualitative analysis. We download these repositories on Nov 11, 2018. As of November 2018, the Openstack organization made 1,253 repositories available. Of these 1,253 repositories we collect repositories for which 11% of all files are IaC scripts. We apply this criterion, because we wanted to collect a large collection of Ansible and Chef scripts so that we have sufficient amount Ansible and Chef code to investigate. All these repositories are hosted on Openstack's public repository browser[9] and not on GitHub.

We provide summary statistics in Table 2 of the 16 Ansible and 10 Chef OSS repositories. The "IaC Cnt" and "IaC Size," respectively, present the total count of IaC scripts and total size of all collected IaC scripts as measured by lines of code.

Methodology Overview: The security smell derivation process is similar to our prior work [37] and the same for all three languages: Ansible, Chef, and Puppet. First, we collect scripts from an organization who have made their scripts available open source. Next, raters with software security knowledge apply open coding to identify coding patterns that satisfy our definition of security smells. Next, we isolate such coding anti-patterns and assign a category. After assigning a category we check for the CWE database. If a mapping is found, then we keep the category;

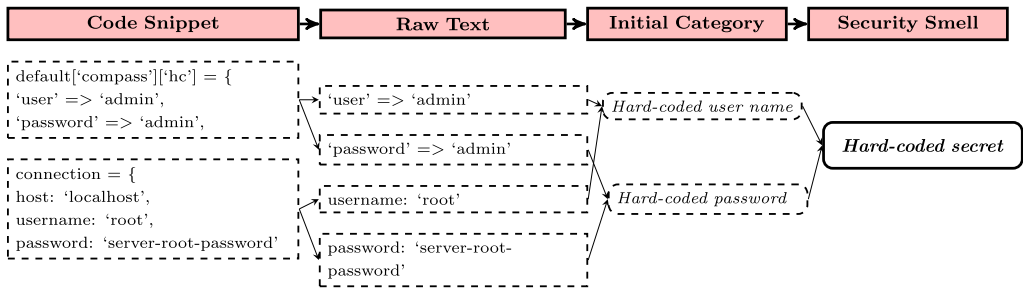
[9]https://git.openstack.org/cgit.

Fig. 4. An example to demonstrate the process of determining security smells using open coding.

otherwise we discard the category. As we use look for coding patterns and CWE, our security smell categorization process can be applied to any configuration language.

Open coding: We first apply a qualitative analysis technique called open coding [43] on the collected scripts. In open coding a rater observes and synthesizes patterns within structured or unstructured text [43]. We select qualitative analysis, because we can (i) get a summarized overview of recurring coding patterns that are indicative of security weakness and (ii) obtain context on how the identified security smells can be automatically identified. We determine security smells by first identifying code snippets that may have security weaknesses based on the first and second author's security expertise. Figure 4 provides an example of our qualitative analysis process. We first analyze the code content for each IaC script and extract code snippets that correspond to a security weakness, as shown in Figure 4. From the code snippet provided in the top left corner, we extract the raw text: 'user' => 'admin'. Next, we generate the initial category "Hard-coded user name" from the raw text "user' => 'admin" and 'username: 'root." Finally, we determine the smell "Hard-coded secret" by combining initial categories. We combine these two initial categories, as both correspond to a common pattern of specifying user names and passwords as hard-coded secrets.

Upon derivation of each smell, we map each identified smell to a possible security weakness defined by CWE [27]. We select the CWE to map each smell to a security weakness, because CWE is a list of common software security weaknesses developed by the security community [27]. A mapping between a derived security smell and a security weakness reported by CWE can validate our qualitative process. For the example presented in Figure 4, we observe the derived security smell "Hard-coded secret" to be related to "CWE-798: Use of Hard-coded Credentials" and "CWE-259: Use of Hard-coded Password" [27]. Each rater separately mapped each of the identified security smell to an entry in the CWE dictionary.

During the time period of conducting open coding, the first author was a PhD student and also the first author of the prior work [37] we replicate. The second author is a PhD student. Both, the first and second author, individually conducted the open coding process. Upon completion of the open coding process, we record the agreements and disagreements for the identified security smells. We also calculate Cohen's Kappa [11].

For Ansible, the first and the second author, respectively, identified four and six security smells. For Chef, the first and the second author respectively identified seven and nine security smells. The Cohen's Kappa is respectively 0.6 and 0.5 for Ansible and Chef scripts between the first and second authors of the article. The disagreements triggered a discussion session, where both raters' reasons on why they agreed or disagreed on the identified smell categories. After completing the discussion, both raters individually revisit their categories, and, finally, both agreed on the set of six and eight security smells, respectively, for Ansible and Chef. At this stage the Cohen's Kappa
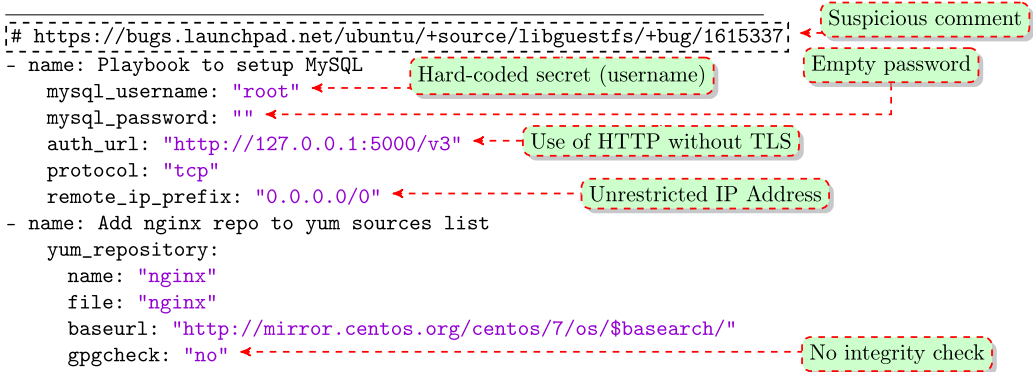
Fig. 5. An annotated Ansible script with six security smells. The name of each security smell is highlighted on the right.

is 1.0 for both Ansible and Chef. One additional security smell for which both raters agreed upon is "No Integrity Check" for both Ansible and Chef.

About Generalizability: Our methodology requires (i) raters with software security experience, (ii) availability of scripts, and (iii) CWE database. As long as these requirements are fulfilled our methodology of deriving smells is generalizable, i.e., can be applied for other IaC languages, such as Terraform. Let us consider a hypothetical example: A researcher wants to replicate our study to derive security smells for Terraform scripts. First step will be using a rater with software security experience. Then, the rater will apply his/her software security knowledge to identify coding patterns and categories. Finally, the rater will check the CWE database if the categories have a direct mapping to the CWE entries.

3.2 Answer to RQ1: What security smells occur in Ansible and Chef scripts?

We identify six security smells for Ansible scripts: empty password, hard-coded secret, no integrity check, suspicious comment, unrestricted IP address, and use of HTTP Without SSL/TLS. For Chef scripts we identify eight security smells: admin by default, hard-coded secret, no integrity check, suspicious comment, switch statement without default, unrestricted IP address, use of HTTP Without SSL/TLS, and use of weak cryptography algorithm. Rahman et al. [37] identified seven security smells for Puppet scripts: admin by default, empty password, hard-coded secret, suspicious comment, unrestricted IP address, use of HTTP Without SSL/TLS, and use of weak cryptography algorithm. Four security smells are common across all of Ansible, Chef, and Puppet: hard-coded secret, suspicious comment, unrestricted IP address, and use of HTTP without SSL/TLS. Examples of each security smell for Ansible and Chef are respectively presented in Figures 5 and 6. Below, we list the names of the smells alphabetically, where each smell name is followed by the applicable language: Ansible (Ⓐ) and Chef (Ⓒ).

Admin by default (Ⓒ): This smell is the recurring pattern of specifying default users as administrative users. The smell can violate the "principle of least privilege" property [30], which recommends practitioners design and implement a system in a manner so that by default the least amount of access necessary is provided to any entity. In Figure 6, an "admin" user will be created in the "default" mode of provisioning an infrastructure. The smell is related with "CWE-250: Execution with Unnecessary Privileges" [27].

Empty password (Ⓐ): This smell is the recurring pattern of using a string of length zero for a password. An empty password is indicative of a weak password. An empty password does not

```
# FIXME: Doesn't work for loop or probably for hp-style          ◀------ Suspicious comment
  default['compass']['hc'] = {          Hard-coded (username), Admin by default
    'user' => 'admin',   ◀------
    'password' => 'admin',   ◀------------------------------------  Hard-coded (password)
    'url' => 'http://127.0.0.1:5000/v2.0',   ◀---------  Use of HTTP without TLS
    'tenant' => 'admin'
  }

  gpgkey 'https://dl.fedoraproject.org/pub/epel/RPM-GPG-KEY-EPEL'   No integrity check
  gpgcheck false   ◀-----------------------------
  method 'md5'   ◀-------------  Use of weak cryptography algorithm
  case node['platform_family']
    when 'suse'
      ip'0.0.0.0'   ◀--------------  Unrestricted IP Address
      package 'xfsdump'
    when 'redhat'
      ip'127.0.0.0'
      package'xfsprogs-devel'   Missing default in case statement
  end   ◀--------------------
```

Fig. 6. An annotated Chef script with eight security smells. The name of each security smell is highlighted on the right.

always lead to a security breach, but makes it easier to guess the password. In SSH key-based authentication, instead of passwords, public and private keys can be used [56]. Our definition of empty password does not include usage of no passwords and focuses on attributes/variables that are related to passwords and assigned an empty string. Empty passwords are not included in hard-coded secrets, because for a hard-coded secret, a configuration value must be a string of length one or more. The smell is similar to the weakness "CWE-258: Empty Password in Configuration File" [27].

Hard-coded secret (Ⓐ Ⓖ): This smell is the recurring pattern of revealing sensitive information, such as user name and passwords in IaC scripts. IaC scripts provide the opportunity to specify configurations for the entire system, such as configuring user name and password, setting up SSH keys for users, specifying authentications files (creating key-pair files for Amazon Web Services). However, programmers can hard-code these pieces of information into scripts. We consider three types of hard-coded secrets: hard-coded passwords, hard-coded user names, and hard-coded private cryptography keys. We acknowledge that practitioners may intentionally leave hard-coded secrets, such as user names and SSH keys in scripts, which may not be enough to cause a security breach. Hence this practice is security smell but not a vulnerability. Relevant weaknesses to the smell are "CWE-798: Use of Hard-coded Credentials" and "CWE-259: Use of Hard-coded Password" [27].

Missing Default in Case Statement (Ⓖ): This smell is the recurring pattern of not handling all input combinations when implementing a case conditional logic. Because of this coding pattern, an attacker can guess a value, which is not handled by the case conditional statements and trigger an error. Such an error can provide the attacker unauthorized information for the system in terms of stack traces or system error. This smell is related to "CWE-478: Missing Default Case in Switch Statement" [27].

No integrity check (Ⓐ Ⓖ): This smell is the recurring pattern of downloading content from the Internet and not checking the downloaded content using checksums or gpg signatures. We observe the following type of content downloaded from the Internet without checking for integrity: .tar, .tgz, .tar.gz, .dmg, .rpm, and .zip. By not checking for integrity, a developer assumes

the downloaded content is secure and has not been corrupted by a potential attacker. Checking for integrity provides an additional layer of security to ensure that the downloaded content is intact, and the downloaded link has not been compromised by an attacker, possibly inserting a virus payload. This smell is related to "CWE-353: Missing Support for Integrity Check" [27].

Suspicious comment (Ⓐ Ⓒ): This smell is the recurring pattern of putting information in comments about the presence of defects, missing functionality, or weakness of the system. Examples of such comments include putting keywords such as "TODO," "FIXME," and "HACK" in comments, along with putting bug information in comments. Keywords such as "TODO" and "FIXME" in comments are used to specify an edge case or a problem [50]. However, these keywords make a comment "suspicious." The smell is related to "CWE-546: Suspicious Comment" [27].

Unrestricted IP Address (Ⓐ Ⓒ): This smell is the recurring pattern of assigning the address 0.0.0.0 for a database server or a cloud service/instance. Binding to the address 0.0.0.0 may cause security concerns as this address can allow connections from every possible network [29]. Such binding can cause security problems as the server, service, or instance will be exposed to all IP addresses for connection. For example, practitioners have reported how binding to 0.0.0.0 facilitated security problems for MySQL[10] (database server), Memcached[11] (cloud-based cache service) and Kibana[12] (cloud-based visualization service). We acknowledge that an organization can opt to bind a database server or cloud instance to 0.0.0.0, but this case may not be desirable overall. This security smell has been referred to as "Invalid IP Address Binding" in our prior work [37]. This smell is related to improper access control as stated in the weakness "CWE-284: Improper Access Control" [27].

Use of HTTP without SSL/TLS (Ⓐ Ⓒ): This smell is the recurring pattern of using HTTP without the Transport Layer Security (TLS) or Secure Sockets Layer (SSL). Such use makes the communication between two entities less secure, as without SSL/TLS, use of HTTP is susceptible to man-in-the-middle attacks [42]. For example, as shown in Figure 5, the authentication URL uses HTTP without SSL/TLS for 'auth_url'. Such usage of HTTP can be problematic, as an attacker can eavesdrop on the communication channel. Information sent over HTTP may be encrypted, and in such case "Use of HTTP without SSL/TLS" may not lead to a security attack. We have referred to this security smell as "Use of HTTP without TLS" in our prior work [37]. This security smell is related to "CWE-319: Cleartext Transmission of Sensitive Information" [27].

Use of weak cryptography algorithms (Ⓒ): This smell is the recurring pattern of using weak cryptography algorithms, namely, MD5 and SHA-1, for encryption purposes. MD5 suffers from security problems, as demonstrated by the Flame malware in 2012 [31]. MD5 is susceptible to collision attacks [12] and modular differential attacks [53]. Similarly to MD5, SHA1 is also susceptible to collision attacks.[13] Using weak cryptography algorithms for hashing that may not always lead to a breach. However, using weak cryptography algorithms for setting up passwords may lead to a breach. This smell is related to "CWE-327: Use of a Broken or Risky Cryptographic Algorithm" and "CWE-326: Inadequate Encryption Strength" [27].

4   SECURITY LINTER FOR ANSIBLE AND CHEF SCRIPTS (SLAC)

We construct SLAC to help practitioners automatically identify security smells in Ansible and Chef scripts. We first describe how we constructed SLAC, and then we describe how we evaluated SLAC's smell detection accuracy.

---

[10]https://serversforhackers.com/c/mysql-network-security.
[11]https://news.ycombinator.com/item?id=16493480.
[12]https://www.elastic.co/guide/en/kibana/5.0/breaking-changes-5.0.html.
[13]https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html.

Line#	Output of Parser
1	<COMMENT, 'This is an example Ansible script'>
2	<KEY, 'name', 'install docker'>
3	<KEY, 'package', '{'name', 'gpgcheck'}'>
4	<KEY, 'name', 'python3'>
5	<KEY, 'gpgcheck', 'false'>

```
1  #This is an example Ansible script
2  − name: install docker
3  − package:
4      name: python3
5      gpgcheck: false
```

a   b

Fig. 7. Output of the "Parser" component in SLAC. (a) An example Ansible script fed to Parser. (b) The output of Parser for the example Ansible script.

```
1  #This is an example Chef script
2  tempVar = 1
3  file "/tmp/test.txt" do
4    content "Test file."
5    owner "test"
6    group "test"
7    mode "00600"
8  end
```

Line#	Output of Parser
1	<COMMENT, 'This is an example Chef script'>
2	<VARIABLE, 'tempVar', 1>
3	<RESOURCE, 'file', '/tmp/test.txt'>
4	<PROPERTY, 'content', 'Test file'>
5	<PROPERTY, 'owner', 'test'>
6	<PROPERTY, 'group', 'test'>
7	<PROPERTY, 'mode, '00600'>

a   b

Fig. 8. Output of the "Parser" component in SLAC. (a) An example Chef script fed to Parser. (b) The output of Parser for the example Chef script.

4.1 Description of SLAC

SLAC is a static analysis tool for detecting the six and eight security smells, respectively, for Ansible and Chef scripts. SLAC has two extensible components:

Parser: The Parser parses an Ansible or Chef script and returns a set of tokens. Tokens are non-whitespace character sequences extracted from IaC scripts, such as keywords and variables. Except for comments, each token is marked with its name, token type, and any associated configuration value. Only token type and configuration value are marked for comments. For example, Figures 7(a) and 8(a) respectively provide a sample script in Ansible and Chef that is fed into SLAC. The output of Parser is expressed as a vector, as shown in Figures 7(b) and 8(b). For example in Figure 8(b), the comment in line#1, is expressed as the vector '<COMMENT, 'This is an example Chef script'>'.

In the case of Ansible, Parser first identifies comments. Next, for non-commented lines Parser uses a YAML parser and constructs a nested list of key-values pairs in JSON format. We use these key-value pairs to construct rules for the Rule Engine.

Similar to Ansible, in the case of Chef, Parser first identifies comments. Next, Parser identifies each token in a Chef script is marked with its name, token type, and any associated configuration value. For example, Figure 8(a) provides a sample script that is fed into SLAC. The output of Parser is is expressed as a vector, as shown in Figure 8(b). For example, the comment in line#1, is expressed as the vector '<COMMENT, 'This is an example Chef script'>'. Parser provides a vector representation of all code snippets in a script.

Rule Engine: Following the study design of prior work [37], we use a rule-based approach to detect security smells. We use rules because (i) unlike keyword-based searching, rules are less susceptible to false positives [51]; and (ii) rules can be applicable for IaC tools irrespective of their

Table 3. An Example of Using Code Snippets to Determine Rule for "Use of HTTP Without SSL/TLS"

Code Snippets	Output of Parser
repo='http://ppa.launchpad.net/chris-lea/node.js-legacy/ubuntu'	<VARIABLE, 'repo', 'http://ppa.launchpad.net/chris-lea/node.js-legacy/ubuntu'>
repo='http://ppa.launchpad.net/chris-lea/node.js/ubuntu'	<VARIABLE, 'repo', 'http://ppa.launchpad.net/chris-lea/node.js/ubuntu'>
auth_uri='http://localhost:5000/v2.0'	<VARIABLE, 'auth_uri', 'http://localhost:5000/v2.0'>
uri 'http://binaries.erlang-solutions.com/debian'	<PROPERTY, 'uri', 'http://binaries.erlang-solutions.com/debian'>
url 'http://pkg.cloudflare.com'	<PROPERTY, 'url', 'http://pkg.cloudflare.com'>

syntax. The Rule Engine consists of a set of rules that correspond to the set of security smells identified in Section 3.1. The Rule Engine uses the set of tokens extracted by Parser and checks if any rules are satisfied.

We can identify properties of source code from the smell-related code snippets and constitute rules using the source code properties. Each smell-related code snippet can show what properties of a script is related with a security smell occurrence. We use Table 3 to demonstrate our approach. The "Code Snippet" column presents a list of code snippets related to "Use of HTTP without SSL/TLS." The "Parser Output" column represents vectors for each code snippet. We observe that the vector of format '<VARIABLE, NAME, CONFIGURATION VALUE >' and '<PROPERTY, NAME, CONFIGURATION VALUE >', respectively, occurs three times and twice for our example set of code snippets. We use the vectors from the output of "Parser" to determine that variable and properties are related to "Use of HTTP without SSL/TLS." The vectors can be abstracted to construct the following rule: '$(isVariable(x) \lor isProperty(x)) \land isHTTP(x)$'. This rule states that "for an IaC script, if token x is a variable or a property, and a string is passed as configuration value for a variable or a property that is related to specifying a URL that uses HTTP without SSL/TLS support, then the script contains the security smell 'Use of HTTP without SSL/TLS'." We apply the process of abstracting patterns from smell-related code snippets to determine the rules for the all security smells for both Ansible and Chef.

A programmer can use SLAC to identify security smells for one or multiple Ansible and Chef scripts. The programmer specifies a directory where script(s) reside. Upon completion of analysis, SLAC generates a comma separated value (CSV) file where the count of security smell for each script is reported. We implement SLAC using API methods provided by PyYAML[14] for Ansible and Foocritic[15] for Chef.

Rules to Detect Security Smells: For Ansible and Chef we present the rules needed for the "Rule Engine" of SLAC respectively in Tables 4 and 5. The string patterns needed to support the rules in Tables 4 and 5 are listed in Table 6. The "Rule" column lists rules for each smell that is executed by Rule Engine to detect smell occurrences. To detect whether or not a token type is a resource ($isResource(x)$), a property ($isProperty(x)$), or a comment ($isComment(x)$), we use the token vectors generated by Parser. Each rule includes functions whose execution is dependent on matching of string patterns. We apply a string pattern-based matching strategy similar to prior work [7, 8], where we check if the value satisfies the necessary condition. Table 6 lists the functions and corresponding string patterns. For example, function 'hasBugInfo()' will return true if the string pattern 'show_bug\.cgi?id=[0-9]+' or 'bug[#\t]*[0-9]+' is satisfied.

---

[14]https://pyyaml.org/.
[15]http://www.foodcritic.io/.

Table 4. Rules to Detect Security Smells for Ansible Scripts

Smell Name	Rule
Empty password	$(isKey(k) \land length(k.value) == 0 \land isPassword(k))$
Hard-coded secret	$(isKey(k) \land length(k.value) > 0) \land$
	$(isUser(k) \lor isPassword(k) \lor isPrivateKey(k))$
No integrity check	$(isKey(k) \land$
	$(isIntegrityCheck(x) == False \land isDownload(x.value)))$
Suspicious comment	$(isComment(k) \land hasWrongWord(k) \lor hasBugInfo(k))$
Unrestricted IP address	$(isKey(k) \land isInvalidBind(k.value))$
Use of HTTP without SSL/TLS	$(isKey(k) \land isHTTP(k.value))$

Table 5. Rules to Detect Security Smells for Chef Scripts

Smell Name	Rule
Admin by default	$(isPropertyOfDefaultAttribute(x)) \land$
	$((isAdmin(x.name)) \land (isUser(x.name) \lor isRole(x.name)))$
Hard-coded secret	$(isProperty(x) \lor isVariable(x)) \land (isUser(x.name) \lor$
	$isPassword(x.name) \lor isPvtKey(x.name)) \land$
	$(length(x.value)>0)$
Missing default in case	$(isCaseStmt(x) \land x.elseBranch == False)$
No integrity check	$(isProperty(x) \lor isAttribute(x)) \land$
	$(isIntegrityCheck(x) == False \land isDownload(x.value))$
Suspicious comment	$(isComment(x)) \land (hasWrongWord(x) \lor hasBugInfo(x))$
Unrestricted IP address	$((isVariable(x) \lor isProperty(x)) \land (isInvalidBind(x.value))$
Use of HTTP without SSL/TLS	$(isProperty(x) \lor isVariable(x)) \land (isHTTP(x.value))$
Use of weak crypto. algo.	$(isAttribute(x) \land usesWeakAlgo(x.value))$

Table 6. String Patterns Used for Functions in Rules

Function	String Pattern
$hasBugInfo()$ [49]	'bug[#\t]*[0-9]+','show_bug\.cgi?id=[0-9]+'
$hasWrongWord()$ [27]	'bug', 'hack', 'fixme', 'later', 'later2', 'todo'
$isAdmin()$	'admin'
$isDownload()$	'http[s]?://(?:[a-zA-Z]|[0-9]|[\$-_@.&+]|[!*,]|(?:%[0-9a-fA-F][0-9a-fA-F]))+.[dmg|rpm|tar.gz|tgz|zip|tar]'
$isHTTP()$	'http:'
$isInvalidBind()$	'0.0.0.0'
$isIntegrityCheck()$	'gpgcheck', 'check_sha', 'checksum', 'checksha'
$isPassword()$	'pwd', 'pass', 'password'
$isPvtKey()$	'[pvt|priv]+*[cert|key|rsa|secret|ssl]+'
$isRole()$	'role'
$isUser()$	'user'
$usesWeakAlgo()$	'md5', 'sha1'

For Ansible and Chef scripts the rule engine takes output from the Parser, and checks if any of the rules listed in Tables 4 and 5, respectively, for Ansible and Chef. In Tables 4 and 5 the "Rule" column lists rules for each smell that is executed by Rule Engine to detect smell occurrences. In the case of Ansible scripts, we used the output from Parser to obtain the key value pairs (k, $k.value$) and comments needed to execute the rules listed in Table 4. Similarly, in the case of Chef scripts, we use the output of Parser to check variables ($isVariable(x)$), properties ($isProperty(x)$), attributes ($isAttribute(x)$), and case statements ($isCaseStmt(x)$). Each rule includes functions whose execution is dependent on matching of string patterns.

4.2 Evaluation of SLAC

We use raters to construct the oracle dataset to mitigate author bias in SLAC's evaluation, similar to Chen et al. [10] and our prior work [37]. We construct four oracle datasets in two rounds. In the first round we use graduate students from NC State University in March and April 2019. In the second round we use a third year Ph.D. student from Tennessee Technological University to construct oracle datasets for Ansible and Chef. In the second round, we do not include Ansible and Chef scripts that are included and analyzed in the first round. We describe oracle dataset construction process for both round in the following subsections:

4.2.1 Evaluation of SLAC in Round#1. We first provide the process of oracle data construction. Next, we provide the performance of SLAC.

Round#1-Oracle Dataset for Ansible and Chef: For each of Ansible and Chef, we construct an oracle dataset using closed coding [43], where at least two raters identifies a pre-determined pattern, and their agreement is checked.

We used graduate students as raters to construct the oracle dataset. We recruited these raters from a graduate-level course related to DevOps conducted in March and April of 2019 at NC State University. Of the 60 students in the class, 32 students agreed to participate. The raters apply their knowledge related to IaC scripts and software security to determine if a certain smell appears for a script. We assigned 96 Ansible and 76 Chef scripts scripts to the 32 students to ensure each script is reviewed by at least two students. The scripts were selected randomly from the 16 Ansible and 10 Chef repositories, respectively, for Ansible and Chef. Each student did not have to rate more than 15 scripts. Prior to allocating the assignments to the students, we obtained Institutional Review Board (IRB) approval (IRB# 12563).

We made the smell identification task available to the raters using a website.[16] The website includes a handbook on Ansible and Chef, and a document that shows examples of security smell instances for both Ansible and Chef. In each task, a rater determines that of the six and eight security smells identified in Section 3.1 occur, respectively, for Ansible and Chef scripts. The graduate students may miss instances of security smells. To mitigate this limitation, after the students conducted closed coding, the first author conducted manual analysis of the 96 Ansible and 76 Chef scripts to identify if security smells have been missed by the raters.

We used balanced block design, a technique to randomly allocate items among multiple categories [6], to assign 96 Ansible and 76 Chef scripts. For Ansible, we observe agreements on the rating for 64 of 96 scripts (66.7%), with a Cohen's Kappa of 0.4. For Chef, we observe agreements on the rating for 61 of 76 scripts (80.2%), with a Cohen's Kappa of 0.5. According to Landis and Koch's interpretation [24], the reported agreement is "fair" and "moderate," respectively, for Ansible and Chef.

---

[16]http://13.59.115.46/website/start.php.

Table 7. SLAC's Accuracy for the Ansible Oracle Dataset (Round#1)

Smell Name	Occurr.	Precision	Recall
Empty password	1	1.0	1.0
Hard-coded secret	1	1.0	1.0
No Integrity Check	2	1.0	1.0
Suspicious comment	4	1.0	1.0
Unrestricted IP address	2	1.0	1.0
Use of HTTP without SSL/TLS	14	1.0	1.0
No smell	77	1.0	1.0
Average		1.0	1.0

Table 8. SLAC's Accuracy for the Chef Oracle Dataset (Round#1)

Smell Name	Occurr.	Precision	Recall
Admin by default	2	1.0	1.0
Hard-coded secret	25	0.8	1.0
Suspicious comment	10	1.0	1.0
Unrestricted IP address	1	1.0	1.0
Use of HTTP without SSL/TLS	27	1.0	1.0
Use of weak crypto. algo.	2	1.0	1.0
No smell	61	1.0	0.9
Average		0.9	0.9

After quantifying the agreement rate, the first author manually inspected 64 Ansible and 61 Chef scripts, for which students agreed. During the manual inspection process, the first author did not use SLAC to identify security smell occurrences. The first author found 17 and 41 security smell occurrences missed by the students, respectively, for Ansible and Chef. The first author added the 17 Ansible and 41 Chef security smells occurrences to the oracle dataset.

Next, the first author resolved disagreements for 32 Ansible scripts and 15 Chef scripts. The disagreements amongst raters occurred for two reasons: (i) students disagreed on the category, and (ii) students disagreed on presence of security smells. After resolving disagreements, and inspecting scripts for which students agreed upon, we obtain an oracle of 24 Ansible and 67 Chef smell occurrences, as listed in the "Occu." columns of Tables 7 and 8. Of the 24 Ansible and 67 Chef smell occurrences, respectively, 7 and 26 smells were identified by the students.

Upon completion of the oracle dataset, we run SLAC for the oracle dataset. Next, we evaluate the accuracy of SLAC using precision and recall for the oracle dataset. Precision refers to the fraction of correctly identified smells among the total identified security smells, as determined by SLAC. Recall refers to the fraction of correctly identified smells that have been retrieved by SLAC over the total amount of security smells.

Round#1-Performance of SLAC for Ansible and Chef Oracle Dataset: We report the detection accuracy of SLAC with respect to precision and recall for Ansible in Table 7 and Chef in Table 8. As shown in the "No smell" row, we identify 77 Ansible scripts with no security smells. The detection accuracy in Tables 7 and 8 corresponds to the accuracy of detecting security smell instances. Along with reporting SLAC's detection accuracy for the oracle dataset, we also report SLAC's detection accuracy for the 7 and 26 security smells identified by the students respectively in Tables 9 and 10.

Table 9. SLAC's Accuracy for Ansible Security Smell Occurrences
Identified Only by Students

Smell Name	Occurr.	Precision	Recall
Suspicious comment	1	1.0	1.0
Use of HTTP without SSL/TLS	6	1.0	1.0
Average		1.0	1.0

Table 10. SLAC's Accuracy for Ansible Security Smell Occurrences
Identified Only by Students

Smell Name	Occurr.	Precision	Recall
Hard-coded secret	5	1.0	1.0
Suspicious comment	9	1.0	1.0
Use of HTTP without SSL/TLS	11	1.0	1.0
Use of weak crypto. algo.	1	1.0	1.0
Average		1.0	1.0

Table 11. SLAC's Accuracy for the Ansible Oracle Dataset (Round#2)

Smell Name	Occurr.	Precision	Recall
Empty password	2	1.0	1.0
Hard-coded secret	18	0.89	1.0
No Integrity Check	8	0.75	1.0
Suspicious comment	10	1.0	1.0
Use of HTTP without SSL/TLS	4	1.0	1.0
No smell	75	1.0	1.0
Average		0.9	1.0

For Tables 9 and 10, students disagreed upon security smell occurrences and categories. The disagreements were resolved by the first author.

4.2.2 Evaluation of SLAC in Round#2. We describe the oracle dataset construction and SLAC's evaluation for the oracle dataset in Round#2.

Round#2-Oracle Dataset for Ansible and Chef: We use a rater who volunteered for constructing the oracle dataset in Round#2. The rater is a third year PhD student at Tennessee Technological University, with a three year experience in software security that included experience in studying vulnerabilities and security bug reports. Similar to round#1, the first author performed additional inspection of the 100 scripts used in round#2.

As shown in Tables 11 and 12, the rater identify 42 and 55 occurrences of security smells respectively for Ansible and Chef scripts. The first author did not find any security smell instances missed by the rater.

Round#2-Performance of SLAC for Ansible and Chef Oracle Dataset: We provide SLAC's evaluation performance for Ansible and Chef respectively in Tables 11 and 12 for Round#2. Evaluation results of SLAC for the oracle datasets is consistent with the evaluation results in Round#1. For Ansible we observe decrease in average precision, but not for average recall. For Chef the average precision and recall is same as in round#1.

Table 12. SLAC's Accuracy for the Chef Oracle Dataset (Round#2)

Smell Name	Occurr.	Precision	Recall
Admin by default	5	1.0	1.0
Hard-coded secret	10	0.75	0.75
Suspicious comment	20	1.0	1.0
Unrestricted IP address	6	1.0	1.0
Use of HTTP without SSL/TLS	7	1.0	1.0
Missing default	9	0.89	1.0
No smell	71	1.0	0.9
Average		0.9	0.9

Dataset and Tool Availability: The source code of SLAC and all constructed datasets are available online [39].

5 EMPIRICAL STUDY

Using SLAC, we conduct an empirical study to quantify the prevalence of security smells in Ansible and Chef scripts.

5.1 Datasets

We conduct our empirical study with four datasets: two datasets each for Ansible and Chef scripts. We construct two datasets from repositories maintained by Openstack. The other two datasets are constructed from repositories hosted on GitHub. We select repositories from Openstack, because Openstack create cloud-based services, and could be a good source for IaC scripts. We include repositories from GitHub, because IT organizations host their OSS projects on GitHub [2, 22]. In contrary to our prior research [37], we only used Openstack datasets as Openstack have made their Ansible and Chef scripts available for download. Ansible and Chef scripts are not available for other organizations, such as Mozilla and Wikimedia.

As advocated by prior research [28], OSS repositories need to be curated. We apply the following criteria to curate the collected repositories:

- **Criterion-1:** At least 11% of the files are IaC scripts. Prior research [18] reported that in OSS repositories IaC scripts co-exist with other types of files, such as Makefiles. A repository that contaisn a few IaC scripts may not be sufficient for analysis. A median of 11% of the files to be IaC scripts was observed [18]. By using a cutoff of 11%, we assume to collect repositories that contain sufficient amount of IaC scripts for analysis.
- **Criterion-2:** The repository is not a clone.
- **Criterion-3:** The repository must have at least two commits per month. We use this criterion to identify repositories with frequent activity. Munaiah et al. [28] used the threshold of at least two commits per month to determine which repositories have enough software development activity.
- **Criterion-4:** The repository has at least 10 developers. Our assumption is that the criteria of at least 10 developers may help us to filter out repositories with limited development activity. Previously, researchers have used the cutoff of at least nine developers [2, 33].

We answer RQ2 using 14,253 Ansible and 36,070 Chef scripts, respectively, collected from 365 and 448 repositories. Table 13 summarizes how many repositories are filtered out using our

Table 13. OSS Repositories Satisfying Criteria (Section 5.1)

	Ansible		Chef	
	GH	OST	GH	OST
Initial Repo Count	3,405,303	1,253	3,405,303	1,253
Criterion-1 (11% IaC Scripts)	13,768	16	5,472	15
Criterion-2 (Not a Clone)	10,017	16	3,567	11
Criterion-3 (Commits/Month ≥ 2)	10,016	16	3,565	11
Criterion-4 (Devs ≥ 10)	349	16	438	10
Final Repo Count	349	16	438	10

Table 14. Summary Attributes of the Datasets

	Ansible		Chef	
Attribute	**GH**	**OST**	**GH**	**OST**
Repository Count	349	16	438	10
Total File Count	498,752	4,487	126,958	2,742
Total Script Count	13,152	1,101	35,132	938
Tot. LOC (IaC Scripts)	602,982	52,239	1,981,203	63,339

criteria. We clone the master branches of these repositories. Summary attributes of the collected repositories are available in Table 14.

5.2 Analysis

Sanity Check: As reported in Section 4.2, SLAC has a high precision and recall for the oracle dataset, but it may under-perform for scripts not included in our oracle dataset. We mitigate this limitation by creating sanity check datasets for 100 Ansible and Chef scripts each that are not included in the oracle dataset. We select these 100 Ansible and 100 Chef scripts randomly form the Openstack dataset constructed in Section 5.1. The first author performs sanity check analysis.

For Ansible we observe 20 scripts to contain at least one security smell. SLAC identifies 45, 1, 36, 2, 16, and 9 occurrences of hard-coded secrets, empty passwords, HTTP without TLS usages, unrestricted IP address bindings, suspicious comments, and no integrity checks. Precision of SLAC for hard-coded secrets, empty passwords, HTTP without TLS usage, unrestricted IP address bindings, suspicious comments, and no integrity checks, respectively, is 0.7, 1.0, 1.0, 1.0, 1.0, and 0.7. Recall of SLAC for hard-coded secrets, empty passwords, HTTP without TLS usage, unrestricted IP address bindings, suspicious comments, and no integrity checks, respectively, is 1.0, 1.0, 1.0, 1.0, 1.0, and 0.9.

For Chef we observe 19 scripts to contain at least one security smell. SLAC identifies 26, 38, 4, 9, 2, and 4 occurrences of hard-coded secrets, HTTP without TLS usage, unrestricted IP address bindings, suspicious comments, missing default in case instances, and no integrity checks. Precision of SLAC for hard-coded secrets, HTTP without TLS usage, unrestricted IP address bindings, suspicious comments, missing default in case instances, and no integrity checks, respectively, is 0.8, 1.0, 1.0, 1.0, 1.0, and 0.8. Recall of SLAC for hard-coded secrets, HTTP without TLS usages, unrestricted IP address bindings, suspicious comments, missing default in case instances, and no integrity checks, respectively, is 1.0, 1.0, 1.0, 1.0, 1.0, and 0.9.

We observe SLAC to generate false positives, but the recall is >= 0.9 for all security smell categories. SLAC's detection accuracy provides confidence on identifying security smells in other scripts not included in the oracle dataset.

5.2.1 Answer to RQ2: How frequently do security smells occur for Ansible and Chef scripts? First, we apply SLAC to determine the security smell occurrences for each script. Second, we calculate two metrics described below:

- **Smell Density:** We use smell density to measure the frequency of a security smell x, for every 1,000 lines of code (LOC). Our smell density metric is similar to that of prior research that have used defect density [20], and is measured using Equation (1).

$$
\text{Smell Density } (x) = \frac{\text{Total occurrences of } x}{\text{Total line count for all scripts}/1000}. \tag{1}
$$

- **Proportion of Scripts (Script%):** We use the metric "Proportion of Scripts" to quantify how many scripts have at least one security smell. This metric refers to the percentage of scripts that contain at least one occurrence of smell x.

5.2.2 RQ3: How do practitioners perceive the identified security smell occurrences for Ansible and Chef scripts? We gather feedback using bug reports on how practitioners perceive the identified security smells. We apply the following procedure:

First, we randomly select 500 occurrences of security smells for each of Ansible and Chef scripts. Second, we post a bug report for each occurrence, describing the following items: smell name, brief description, related CWE, and the script where the smell occurred. We explicitly ask if contributors of the repository agrees to fix the smell instances. Third, we determine a practitioner to agree with a security smell occurrence if (i) the practitioner replies to the submitted bug report explicitly saying the practitioner agrees or (ii) the practitioner fixes the security smell occurrence in the specified script by running SLAC on IaC scripts, for which we submitted bug reports. If the security smell does not exist in the script of interest, then we determine the smell to be fixed.

6 EMPIRICAL FINDINGS

We answer RQ2 and RQ3 in this section.

6.1 Answer to RQ2: How frequently do security smells occur for Ansible and Chef scripts?

We observe our identified security smells to exist across all datasets. For Ansible, in our GitHub and Openstack datasets we observe respectively 25.3% and 29.6% of the total scripts to contain at least one of the six identified security smells. For Chef, in our GitHub and Openstack datasets we observe respectively 20.5% and 30.4% of the total scripts to contain at least one of the eight identified security smells. A complete breakdown of findings related to RQ2 for Ansible and Chef is presented in Tables 15, 16, and 17 for our datasets.

Occurrences: The occurrences of the security smells are presented in the "Occurrences" column of Table 15 for all datasets. The "Combined" row presents the total smell occurrences. In the case of Ansible scripts, we observe 18,353 occurrences of security smells, and for Chef, we observe 28,247 occurrences of security smells. For Ansible, we identify 15,131 occurrences of hard-coded secrets, of which 55.9%, 37.0%, and 7.1% are, respectively, hard-coded keys, user names, and passwords. For Chef, we identify 15,363 occurrences of hard-coded secrets, of which 47.0%, 8.9%, and 44.1% are, respectively, hard-coded keys, user names, and passwords.

Table 15. Smell Occurrences for Ansible and Chef Scripts

	Ansible		Chef	
Smell Name	**GH**	**OST**	**GH**	**OST**
Admin by default	N/A	N/A	301	61
Empty password	298	3	N/A	N/A
Hard-coded secret	14,409	722	14,160	1,203
Missing default in switch	N/A	N/A	953	68
No integrity check	194	14	2,249	132
Suspicious comment	1,421	138	3,029	161
Unrestricted IP address	129	7	591	19
Use of HTTP without SSL/TLS	934	84	4,898	326
Use of weak crypto algo.	N/A	N/A	94	2
Combined	17,385	968	26,275	1,972

Table 16. Smell Density for Ansible and Chef Scripts

	Ansible		Chef	
Smell Name	**GH**	**OST**	**GH**	**OST**
Admin by default	N/A	N/A	0.1	0.9
Empty password	0.49	0.06	N/A	N/A
Hard-coded secret	23.9	13.8	7.1	19.0
Missing default in switch	N/A	N/A	0.5	1.0
No integrity check	0.3	0.2	1.1	2.1
Suspicious comment	2.3	2.6	1.5	2.5
Unrestricted IP address	0.2	0.1	0.3	0.3
Use of HTTP without SSL/TLS	1.5	1.6	2.4	5.1
Use of weak crypto algo.	N/A	N/A	0.05	0.03
Combined	28.8	18.5	13.3	31.5

Table 17. Proportion of Scripts with at Least One Smell for Ansible and Chef Scripts

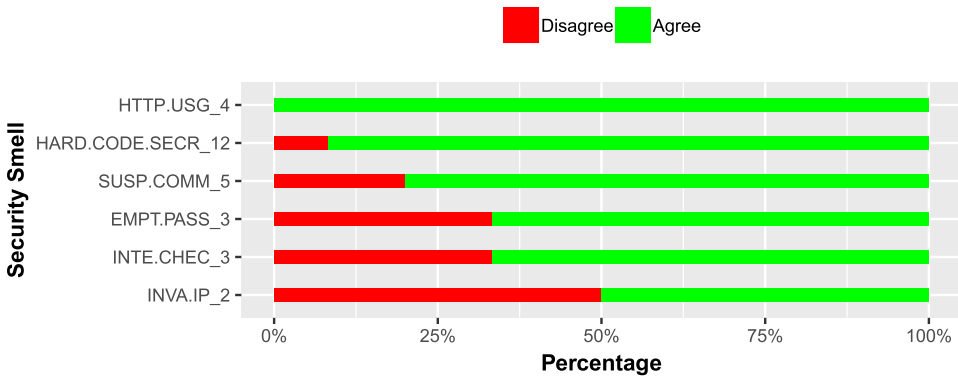|	Ansible		Chef	
Smell Name	**GH**	**OST**	**GH**	**OST**
Admin by default	N/A	N/A	0.3	2.1
Empty password	1.1	0.2	N/A	N/A
Hard-coded secret	19.2	22.4	6.8	15.9
Missing default in switch	N/A	N/A	2.5	6.5
No integrity check	1.1	1.0	3.6	3.8
Suspicious comment	6.3	8.0	6.6	9.3
Unrestricted IP address	0.5	0.4	1.1	1.0
Use of HTTP without SSL/TLS	3.7	3.0	4.9	6.9
Use of weak crypto algo.	N/A	N/A	0.2	0.1
Combined	25.3	29.6	20.5	30.4

Fig. 9. Feedback for 29 smell occurrences for Ansible. Practitioners agreed with 82.7% of the selected smell occurrences.

Exposing hard-coded secrets, such as hard-coded keys, is not uncommon: Meli et al. [26] studied secret key exposure in OSS GitHub repositories, and identified 201,642 instances of private keys, which included commonly-used API keys. Meli et al. [26] reported 85,311 of the identified 201,642 instances of private keys to be Google API keys.

Smell Density: In Table 16, we report the smell density for both, Ansible and Chef. The "Combined" row presents the smell density for each dataset when all identified security smell occurrences are considered. For all datasets, we observe the dominant security smell to be "Hard-coded secret."

Proportion of Scripts (Script%): In Table 17, we report the proportion of scripts (Script %) values for each of the four datasets. The "Combined" row represents the proportion of scripts in which at least one of the identified smells appear.

6.2 Answer to RQ3: How do practitioners perceive the identified security smell occurrences for Ansible and Chef scripts?

From 7 and 30 repositories, respectively, we obtain 29 and 65 responses for the submitted 500 Ansible and the 500 Chef security smell occurrences. In the case of Ansible, we observe an agreement of 82.7% for 29 smell occurrences. For Chef, we observe an agreement of 63.1% for 65 smell occurrences. The percentage of smells to which practitioners agreed to be fixed for Ansible and Chef is respectively presented in Figure 9 and 10. In the y-axis each smell name is followed by the occurrence count. For example, according to Figure 9, for 4 occurrences of "Use of HTTP without SSL/TLS" (HTTP.USG), we observe 100% agreement for Ansible scripts.

We acknowledge that the response rate is 9.4%, which is low for the submitted bug reports. One possible explanation can be developers might be biased against security smell alerts, as they are typically generated by static analysis tools. Upon submission of the bug reports, developers may have considered the identified security smells as "code smells," and left these bug reports as unresolved. Developers incorrect perceptions on insecure coding is not uncommon: for example, Acar et al. [1] have observed developers bias to perceive their code snippets as secure, even if the code snippets are insecure.

Another possible explanation can be we have submitted bug reports for repositories that are inactive despite the applying systematic criteria to filter the repositories. For example, for one bug report a practitioner mentioned that the repository 'rcbops/ansible-lxc-rpc/' is no longer maintained.[17]

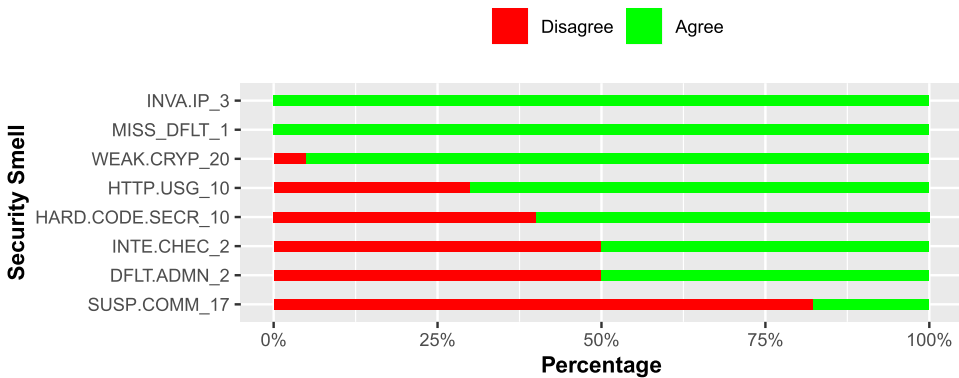[17]https://github.com/rcbops/ansible-lxc-rpc/issues/681.

Fig. 10. Feedback for the 65 smell occurrences for Chef. Practitioners agreed with 63.1% of the selected smell occurrences.

Another possible explanation can be lack of actionability: The submitted bug reports do not provide suggestions on how to act on the security smells. As an example, if a hard-coded password appears in an Ansible or Chef script, we do not discuss what techniques should be adopted to repair the smell occurrence in the bug report.

Reasons for Practitioner Agreement: Lack of awareness and availability of repair suggestions attributed to why practitioner agreed to security smell instances. We provide examples below:

- *Awareness of HTTPS availability*: We submitted a bug report for two instances of "HTTP without SSL/TLS." For both instances a URL was used to download RStudio packages we submitted a bug report.[18] In response to the bug report, the practitioner agreed that the smell instance needs to be repaired, and repaired the smell instances.[19] The practitioner also stated why the smell instance was introduced in the first place: "*In this case, I think it was just me being a bit sloppy: the HTTPS endpoint is available so I should have used that to download RStudio packages from the start.*"

- *Awareness on hard-coded secrets*: For an instance of hard-coded user name and hard-coded password for an Ansible script[20] we submitted a bug report.[21] In response, the practitioner acknowledged the presence of the smells. The practitioner also stated what actions he can take to mitigate the security smells: "*I agree that it [hard-coded secret] could be in an Ansible vault or something dedicated to secret storage.*"

- *Availability of repair suggestions*: For one instance of weak cryptography usage in a Chef script, we submitted a bug report.[22] Along with submitting the bug report, we also submitted a pull request that replaced MD5 usage with SHA512.[23] The pull request was accepted a month later, and presence of the security smell was acknowledged by the practitioner.

Reasons for Practitioner Disagreement: For disagreements we observe development context to be an important factor. We provide examples below:

---

[18] https://github.com/elasticluster/elasticluster/issues/634.

[19] https://github.com/elasticluster/elasticluster/commit/a62b8aae6559a3a15fbb724709005caba8cf33e8.

[20] https://github.com/quarkslab/irma/blob/master/ansible/playbooks/group_vars/all.yml.

[21] https://github.com/quarkslab/irma/issues/60.

[22] https://github.com/cookbooks/hw-postgresql/issues/1.

[23] https://github.com/cookbooks/hw-postgresql/pull/2/commits/66f9841177080988d2af9789f92daa4c0a1b325d.

- *Dependency*: Practitioners may disagree with instances of "use of HTTP without SSL/TLS" for a URL, if the URL refers to a dependency maintained by an external organization, upon which the practitioner or the team has no control upon. For example in a bug report,[24] we observe a practitioner to disagree with occurrences of "use of HTTP without SSL/TLS" in a Chef script.[25] All of these URLs refer to remote archive hosts maintained by Cloudera,[26] an IT organization that provides cloud utilities. The practitioner disagreed and asked to report this issue to "upstream," i.e., to the project maintainers who manage the URLs.

- *Location of smells*: A hard-coded password may not have security implications for practitioners if the hard-coded password is located in testing code of Chef or Ansible scripts. In one bug report[27] a practitioner stated "*the code in question is an integration test. The username and password is not used anywhere else so this should be no issue.*" The practitioners views were echoed for another instance of hard-coded password, which we reported as a bug report.[28] The practitioner also provided suggestions on how we can prioritize inspection: "*I suggest that the author probably needs to adjust his scanner to not be quite so sensitive when it detects usernames and passwords set in RSpec or Inspec code. Or at least to prompt the person running the script before creating an issue on a repository. Human intervention is likely the best principled action, here.*" For both bug reports the practitioners assume that hard-coded usernames and passwords in test code is not relevant as the hard-coded password will never be used in production system. One possible limitation to such assumption is that practitioners are only considering their own development context, and not realizing how another practitioner, not experienced in IaC, may perceive use of these security smells as an acceptable practice. Strong perceptions on researcher submitted bug reports in not uncommon as documented in GitHub issues. Developers have strong perceptions about bugs identified by research tools on whether they are "important" or not. For example, developers of Z3 strongly disagreed on a bug reported by researchers, because the identified bug is "*asinine.*" Furthermore, the developer adds "*As someone who uses Z3/Boolector/STP/CVC4 1000s of times a day, I would much rather that issue trackers such as these are full-up with issues that real users find, than the ones you derive.*"[29]

7 DISCUSSION

We discuss implications of our article as following:

7.1 Towards Actionable Detection and Repair: Lessons Learned

Practitioner responses from the submitted bug reports provide signals on how we can make SLAC more actionable with respect to detection. We have learned that practitioners do not consider hard-coded user names and passwords in testing scripts as relevant. Toolsmiths can take this observation into account and tune future security smell detection tools accordingly. We also learned that URL instances related to "use of HTTP without TLS" might not be relevant if an HTTPS URL exists in the first place. For example, in the case of generating a security smell alert, for http://archive.cloudera.com/debian/archive.key, SLAC could have been adjusted to check for the availability of a secure HTTP endpoint for the URL.

Along with submitting bug reports for the detected static analysis alerts, automated pull requests can be generated that will include repair suggestions for the detected security smell

---

[24]https://github.com/cookbooks/ic-cassandra/issues/2.

[25]https://github.com/cookbooks/ic-cassandra/blob/master/cookbooks/hadoop_cluster/recipes/add_cloudera_repo.rb.

[26]https://www.cloudera.com/.

[27]https://github.com/Graylog2/graylog2-cookbook/issues/109.

[28]https://github.com/chef-cookbooks/docker/issues/1069.

[29]https://github.com/Z3Prover/z3/issues/4461.

instances. Practitioners might be more receptive to a security smell instance, if the alert notification also accompanies suggestions on how to repair. For example, toolsmiths can create tools that will generate automated pull requests, which will show how to repair a security smell instance.

7.2 Additional Contributions Compared to Our Prior Research

As discussed in Section 2.1.2, the three IaC languages, namely, Ansible, Chef, and Puppet differ from each other with respect to execution order, perceived codebase maintenance, requiring additional agent software installation, style, and syntax. The above-mentioned differences merit a systematic investigation of the categorization and quantification of security smells for Ansible and Chef scripts. From a practitioner perspective, if a team only uses Ansible scripts, then the catalog of security smells for Puppet scripts and the tool to detect security smells from our prior work may not be relevant. Similarly, for practitioners using Chef, the security smell catalog and tool for Puppet or Ansible might not be relevant. We have noticed anecdotal evidence related to this: upon completion of our prior work [37], we reached out to practitioners for feedback. One feedback was "*This is practical. Does the tool work for Ansible?*" Our replication study addresses the needs of practitioners who use Ansible and/or Chef.

Schwarz et al. [47] pursued similar efforts for Chef's code maintenance smells research. They [47] replicated Sharma et al. [48]'s research on Puppet code maintenance smells for Chef scripts, and observed Puppet's code smells to appear for Chef as well. Schwarz et al. [47]'s article is an example how replication can benefit the researcher community to advance the science of IaC script quality. We too have followed the footsteps of Schwarz et al. [47] and replicated our prior research for Ansible and Chef.

In short, the differences in contributions between our prior work and this article is following:

- A list of security smells for Ansible and Chef scripts that includes two categories not reported in prior work [37];
- An evaluation of frequency of security smells occur in Ansible and Chef scripts. As a result of this evaluation we have created a benchmark how frequently security smells appear for Ansible and Chef. To date, such a benchmark is missing. Frequency of identified security smells for Ansible and Chef scripts can be used as a measuring stick by practitioners and researchers alike;
- A detailed discussion on how practitioner responses from bug reports can drive actionable detection and repair of Ansible and Chef security smells. In our prior work we did not discuss how the practitioners responses in bug reports can guide tools for actionable detection and repair;
- An empirically validated tool (SLAC) that automatically detects occurrences of security smells for Ansible, and Chef scripts. The tool that we constructed as part of prior work will not work for Ansible and Chef scripts. The "Parser" component of SLAC differs from that of "SLIC" that we built in our prior work. The "Rule Engine" component of SLAC differs from that of SLIC, as, unlike Puppet, which uses attributes, Ansible and Chef respectively use "Keys" and "Properties"; and
- A detailed discussion of differences among the three IaC languages, Ansible, Chef, and Puppet. In our prior work, we provided background on Puppet scripts only and did not discuss the differences among Ansible, Chef, and Puppet.

7.3 Differences in Security Smell Occurrences for Ansible, Chef, and Puppet Scripts

Our identified security smells for Ansible and Chef overlap with Puppet. The security smells that are common across all three languages are: hard-coded secret, suspicious comment, unrestricted

IP address, and use of HTTP Without SSL/TLS. Security smells identified for Puppet are also applicable for Chef and Ansible, which provides further validation to our prior research findings [37]. We also identify additional security smells namely, "Missing Default in Case" and "No Integrity Check," which was not previously identified by Rahman et al. [37]. One possible explanation can be related to rater bias: in our prior work, we used one rater to identify security smells in Puppet scripts. The rater might have missed instances of "Missing Default in Case" and "No Integrity Check." Another possible explanation can be the set of scripts the rater used for inspection in prior work [37]. Perhaps, those set of scripts were carefully developed by developers who are aware of the security consequences related to the new categories.

Despite differences in frequency of security smells across datasets and languages, we observe the proportion of scripts to contain at least one smell varies between 20.5% and 32.9%. Our findings indicate that some IaC scripts, regardless of their languages may include operations that make those scripts susceptible to security smells. Our finding is congruent with Rahman and Williams' observations [40]: They observed defective Puppet scripts to contain certain operations, operations related to filesystem, infrastructure provisioning, and user account management. Based on our findings and prior observations from Rahman and Williams [40] we conjecture that similar to defective scripts, IaC scripts with security smells may also include certain operations that distinguishes them from scripts with no security smells.

Our results related to Ansible and Chef overlap that with prior research [37]. Despite the overlap, our results have implications for practitioners, toolsmiths, and educators. The fact that our research results related to Chef and Ansible overlap the findings for Puppet highlights a lack of awareness related to security for IaC. Regardless of what IaC language is being used, certain security smells, such as hard-coded secrets are dominant. Practitioners who are using Ansible, Chef, or Puppet scripts should be aware of the security consequences. Toolsmiths can build upon our tools, SLIC [37] and SLAC to detect security smells in other IaC languages, such as Terraform. Educators who are teaching DevOps and configuration management should discuss the security implications of security smells in IaC scripts. The commonality of the identified security smells across the three programming languages provides evidence related to robustness of our prior work conducted only on Puppet [37]. Endres et al. [13] suggested that research results confirmed by diverse data sources help advance scientific research in the field of software engineering.

7.4 On the Value of Replication for IaC Research

Presented findings in our article is an example on why IaC-related research should be replicated for other programming languages. We have identified two new security smell categories that were not reported in prior research. For IaC-related research typically researchers have relied on Chef and Puppet [36]. However, structural and semantic differences exist among IaC-related programming languages, and the IaC community may benefit from replication studies, which can identify differences and similarities in research conclusions across multiple programming languages. For example, Rahman and Williams [41]'s work on identifying source code properties can be replicated for a larger set of scripts developed in other languages, such as Ansible. As another example, Hummer et al. [17]'s research on Chef idempotency can be replicated for Ansible and Puppet scripts. In multiple blog posts,[30,31,32] practitioners have mentioned how one language can differ

---

[30]https://www.simplilearn.com/ansible-vs-puppet-the-key-differences-to-know-article.
[31]https://www.devopsgroup.com/blog/puppet-vs-ansible/.
[32]https://blog.gruntwork.io/why-we-use-terraform-and-not-chef-puppet-ansible-saltstack-or-cloudformation-7989dad2865c.

from another with respect to syntax, scalability, and configuration management philosophy.[33] The domain of IaC research can benefit from replication studies where IaC scripts written in two or more languages can be used to confirm or negate research hypotheses.

7.5 Mitigation Strategies

Admin by default: Practitioners can follow the recommendations from Saltzer and Schroeder [44] to create user accounts that have the minimum possible security privilege and use that account as default.

Empty password: The use of strong passwords can mitigate appearance of empty passwords in Ansible and Chef scripts.

Hard-coded secret: We provide two suggestions: First, scan IaC scripts to search for hard-coded secrets using tools such as CredScan[34] and SLAC. Second, use tools such as Ansible/AWS[35] and Vault[36] to store secrets.

Missing default in case statement: We advise programmers to always add a default "else" block so that unexpected input does not trigger events, which can expose information about the system.

No integrity check: As IaC scripts are used to download and install packages and repositories at scale, we advise practitioners to always check downloaded content by computing hashes of the content or checking with GPG signatures.

Suspicious comment: We acknowledge that in OSS development, programmers may be introducing suspicious comments to facilitate collaborative development and to provide clues on why the corresponding code changes are made [50]. Based on our findings we advocate for creating explicit guidelines on what pieces of information to store in comments, and strictly follow those guidelines through code review. For example, if a programmer submits code changes where a comment contains any of the patterns mentioned for suspicious comments in Table 6, the submitted code changes will not be accepted.

Unrestricted IP address: To mitigate this smell, we advise programmers to allocate their IP addresses systematically based on which services and resources need to be provisioned. For example, incoming and outgoing connections for a database containing sensitive information can be restricted to a certain IP address and port.

Use of HTTP without SSL/TLS: We advocate companies to adopt the HTTP with SSL/TLS by leveraging resources provided by tool vendors, such as MySQL[37] and Apache.[38] We advocate for better documentation and tool support so that programmers do not abandon the process of setting up HTTP with SSL/TLS.

Use of weak cryptography algorithms: We advise programmers to use cryptography algorithms recommended by the National Institute of Standards and Technology [5] to mitigate this smell. For example, "MD5" usages should be replaced by "SHA256" or "SHA512."

7.6 Future Work

From Section 6.1, answers to RQ2 indicate that not all IaC scripts include security smells. Researchers can build upon our findings to explore which characteristics correlate with IaC scripts

---

[33]https://www.upguard.com/articles/ansible-puppet.

[34]https://secdevtools.azurewebsites.net/helpcredscan.html.

[35]https://github.com/ansible/awx.

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

[37]https://dev.mysql.com/doc/refman/5.7/en/encrypted-connections.html.

[38]https://httpd.apache.org/docs/2.4/ssl/ssl_howto.html.

with security smells. If certain characteristics correlate with scripts that have smells, then programmers can prioritize their inspection efforts for scripts that exhibit those characteristics. Researchers can investigate how semantics and dynamic analysis of scripts can help in efficient smell detection. Researchers can also investigate what remediation strategies can be adopted that facilitate better actionability and repair of security smells identified by SLAC. As our detection accuracy results indicate, SLAC generates false positives, which can motivate future work to detect security smells with high precision.

We have not quantified lifetime of security smells for Ansible and Chef scripts. Quantifying the lifetime of security smells for Ansible and Chef scripts is an excellent idea, which will require significant change in the design of SLAC. Currently, SLAC detects the presence of security smells in Ansible and Chef scripts. For lifetime detection SLAC should be expanded to handle (i) code snippets where smell appears, (ii) obtaining each version of all 50,323 scripts over 9 years, and (iii) using heuristics to compare code snippets across 9 years that include security smells. Researchers can investigate the lifetime of Ansible and Chef scripts in future.

8 THREATS TO VALIDITY

In this section, we discuss the limitations of our article:

Conclusion Validity: The derived security smells and their association with CWEs are subject to the rater judgment. During the security smell derivation process the first author was involved, who also derived the security smells for Puppet scripts [37]. The first author's bias can influence the smell derivation process for Ansible and Chef scripts. We account for this limitation by using another rater, second author of the article, who is experienced in software security.

The oracle datasets were constructed by the raters. The construction process is susceptible to subjectivity, as the raters' judgment influences appearance of a certain security smell. We mitigate this limitation by allocating at least two raters for each script. We have used graduate students to construct oracle datasets. But as reported in Section 4.2, students miss security smell occurrences. We mitigate this limitation by using the first author who identified security smell instances missed by the graduate students. However, in the process, bias inherent in the first author's judgement can influence the construction of the oracle dataset. We mitigate this limitation by constructing another oracle dataset with a volunteer rater.

We use certain thresholds to curate repositories based on observations reported in prior research [2, 18, 28]. Our selection thresholds can be limiting. For example, a repository may contain sufficient amount of Ansible or Chef scripts, but maintained by one practitioner. Such repositories even though active, will be excluded in our analysis based on criteria mention in Section 5.

Internal Validity: We acknowledge that other security smells may exist for both Ansible and Chef. We mitigate this threat by manually analyzing 1,101 Ansible and 855 Chef scripts for security smells. In the future, we aim to investigate if more security smells exist.

The detection accuracy of SLAC depends on the constructed rules that we have provided in Tables 4 and 5. We acknowledge that the constructed rules are susceptible of generating false positives and false negatives. Accuracy of SLAC is dependent on the string patterns used in Table 6.

External Validity: Our findings are subject to external validity, as our findings may not be generalizable. We observe how security smells are subject to practitioner interpretation, and thus the relevance of security smells may vary from one practitioner to another. Also, our scripts are collected from the OSS domain, and not from proprietary sources. We conduct our investigation with two languages, Ansible and Chef. Investigation of other languages used in IaC, such as Terraform, can reveal new categories of security smells. Also, reported detection accuracy for SLAC is limited to the two oracle datasets and the sanity check dataset.

9 CONCLUSION

IaC is the practice of using automated scripting to provision computing environments by applying recommended software engineering practices, such as version control and testing. Security smells are recurring coding patterns in IaC scripts that are indicative of security weakness and can potentially lead to security breaches. By applying open coding on 1,101 Ansible and 855 Chef scripts, we identified six and eight security smells, respectively, for Ansible and Chef. The security smells that are common across all three languages are: hard-coded secret, suspicious comment, unrestricted IP address, and use of HTTP Without SSL/TLS.

Next, we construct a static analysis tool called SLAC using which we analyzed 50,323 Ansible and Chef scripts. We identify 46,600 security smells by analyzing 50,323 scripts, which included 7,849 hard-coded passwords. Based on smell density, we observed the most dominant smell to be "Hard-coded secret." We observe security smells to be prevalent in Ansible and Chef scripts. We recommend practitioners to rigorously inspect the presence of the identified security smells through code review and by using automated static analysis tools for IaC scripts. We hope our article will facilitate further security-related research in the domain of IaC scripts.

ACKNOWLEDGMENTS

REFERENCES

[1] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky. 2017. Comparing the usability of cryptographic APIs. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP'17)*. 154–171. DOI: https://doi.org/10.1109/SP.2017.52

[2] Amritanshu Agrawal, Akond Rahman, Rahul Krishna, Alexander Sobran, and Tim Menzies. 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 (ICSE-SEIP'18)*. ACM, New York, NY, 245–253. DOI: https://doi.org/10.1145/3183519.3183549

[3] Ansible. 2019. *NASA: Increasing Cloud Efficiency with Ansible and Ansible Tower*. Technical Report. Ansible.

[4] Ansible. 2020. Ansible Project. Retrieved April 25, 2020 from https://docs.ansible.com/.

[5] Elaine Barker. 2016. *Guideline for Using Cryptographic Standards in the Federal Government: Cryptographic Mechanisms*. Technical Report. National Institute of Standards and Technology, Gaithersburg, Maryland.

[6] Raj Chandra Bose. 1939. On the construction of balanced incomplete block designs. *Ann. Eugen.* 9, 4 (1939), 353–399.

[7] Amiangshu Bosu, Jeffrey C. Carver, Munawar Hafiz, Patrick Hilley, and Derek Janni. 2014. Identifying the characteristics of vulnerable code changes: An empirical study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. ACM, New York, NY, 257–268. DOI: https://doi.org/10.1145/2635868.2635880

[8] Sven Bugiel, Stefan Nurnberger, Thomas Poppelmann, Ahmad-Reza Sadeghi, and Thomas Schneider. 2011. Amazon IA: When elasticity snaps back. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*. ACM, New York, NY, 389–400. DOI: https://doi.org/10.1145/2046707.2046753

[9] Chef. 2018. Sitemap-Chef Docs. Retrieved July 4, 2019 from https://docs.chef.io/.

[10] B. Chen and Z. M. Jiang. 2017. Characterizing and detecting anti-patterns in the logging code. In *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE'17)*. 71–81. DOI: https://doi.org/10.1109/ICSE.2017.15

[11] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educ. Psychol. Meas.* 20, 1 (1960), 37–46. DOI: https://doi.org/10.1177/001316446002000104

[12] Bert den Boer and Antoon Bosselaers. 1994. Collisions for the compression function of MD5. In *Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology (EUROCRYPT'93)*. Springer-Verlag, Secaucus, NJ, 293–304. http://dl.acm.org/citation.cfm?id=188307.188356.

[13] Albert Endres and H. Dieter Rombach. 2003. *A Handbook of Software and Systems Engineering: Empirical Observations, Laws, and Theories*. Pearson Education.

[14] Martin Fowler and Kent Beck. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.

[15] Oliver Hanappi, Waldemar Hummer, and Schahram Dustdar. 2016. Asserting reliable convergence for configuration management scripts. *SIGPLAN Not.* 51, 10 (October 2016), 328–343. DOI: https://doi.org/10.1145/3022671.2984000

[16] Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation* (1st ed.). Addison-Wesley Professional.

[17] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. 2013. Testing idempotence for infrastructure as code. In *Proceedings of the Middleware 2013*, David Eyers and Karsten Schwan (Eds.). Springer, 368–388.

[18] Yujuan Jiang and Bram Adams. 2015. Co-evolution of infrastructure and source code: An empirical study. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR'15)*. IEEE Press, Piscataway, NJ, 45–55. http://dl.acm.org/citation.cfm?id=2820518.2820527.

[19] Natalia Juristo and Omar S. Gómez. 2010. Replication of software engineering experiments. In *Empirical Software Engineering and Verification*. Springer, 60–88.

[20] John C. Kelly, Joseph S. Sherif, and Jonathan Hops. 1992. An analysis of defect densities found during software inspections. *J. Syst. Softw.* 17, 2 (1992), 111–117. DOI: https://doi.org/10.1016/0164-1212(92)90089-3

[21] Jonathan L. Krein and Charles D. Knutson. 2010. A case for replication: Synthesizing research methodologies in software engineering. In *Proceedings of the 1st International Workshop on Replication in Empirical Software Engineering Research*.

[22] Rahul Krishna, Amritanshu Agrawal, Akond Rahman, Alexander Sobran, and Tim Menzies. 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 (ICSE-SEIP'18)*. ACM, New York, NY, 306–315. DOI: https://doi.org/10.1145/3183519.3183548

[23] Puppet Labs. 2018. *Borsa Istanbul: Improving Efficiency and Reducing Costs to Manage a Growing Infrastructure*. Technical Report. Puppet.

[24] J. Richard Landis and Gary G. Koch. 1977. The measurement of observer agreement for categorical data. *Biometrics* 33, 1 (1977), 159–174.

[25] Mike Leone. 2016. *The Economic Benefits of Puppet Enterprise*. Technical Report. ESG.

[26] Michael Meli, Matthew R. McNiece, and Bradley Reaves. 2019. How bad can it git? Characterizing secret leakage in public GitHub repositories. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium, (NDSS'19)*.

[27] MITRE. 2018. CWE-Common Weakness Enumeration. Retrieved July 2, 2019 from https://cwe.mitre.org/index.html.

[28] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empir. Softw. Eng.* 22, 6 (2017), 1–35. DOI: https://doi.org/10.1007/s10664-017-9512-6

[29] Pars Mutaf. 1999. Defending against a Denial-of-Service Attack on TCP. In *Recent Advances in Intrusion Detection*.

[30] National Institute of Standards and Technology. 2014. Security and Privacy Controls for Federal Information Systems and Organizations. Retrieved July 4, 2019 from https://www.nist.gov/publications/security-and-privacy-controls-federal-information-systems-and-organizations-including-0.

[31] Laboratory of Cryptography and System Security (CrySyS). 2012. *sKyWIper (a.k.a. Flame a.k.a. Flamer): A Complex Malware for Targeted Attacks*. Technical Report. Laboratory of Cryptography and System Security, Budapest, Hungary.

[32] Puppet. 2018. *Ambit Energy's Competitive Advantage? It's Really a DevOps Software Company*. Technical Report. Puppet.

[33] Akond Rahman, Amritanshu Agrawal, Rahul Krishna, and Alexander Sobran. 2018. 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 (SWAN'18)*. ACM, New York, NY, 8–14. DOI: https://doi.org/10.1145/3278142.3278149

[34] Akond Rahman, Effat Farhana, Chris Parnin, and Laurie Williams. 2020. Gang of eight: A defect taxonomy for infrastructure as code scripts. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE'20)*.

[35] Akond Rahman, Effat Farhana, and Laurie Williams. 2020. The 'as code' activities: Development anti-patterns for infrastructure as code. *Empirical Softw. Engg.* 25, 5 (2020), 43. DOI: https://doi.org/10.1007/s10664-020-09841-8

[36] Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie Williams. 2018. A systematic mapping study of infrastructure as code research. *Inf. Softw. Technol.* 108, 4 (2018). DOI: https://doi.org/10.1016/j.infsof.2018.12.004

[37] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The seven sins: Security smells in infrastructure as code scripts. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*. IEEE Press, Piscataway, NJ, 164–175. DOI: https://doi.org/10.1109/ICSE.2019.00033

[38] Akond Rahman, Asif Partho, David Meder, and Laurie Williams. 2017. Which factors influence practitioners' usage of build automation tools? In *Proceedings of the 3rd International Workshop on Rapid Continuous Software Engineering (RCoSE'17)*. IEEE Press, Piscataway, NJ, 20–26. DOI: https://doi.org/10.1109/RCoSE.2017..8

[39] Akond Rahman, M. Rahman, Chris Parnin, and Laurie Williams. 2020. Dataset for security smells for ansible and chef scripts used in DevOps. DOI : https://doi.org/10.6084/m9.figshare.8085755

[40] A. Rahman and L. Williams. 2018. Characterizing defective configuration scripts used for continuous deployment. In *Proceedings of the 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST'18)*. 34–45. DOI : https://doi.org/10.1109/ICST.2018.00014

[41] Akond Rahman and Laurie Williams. 2019. Source code properties of defective infrastructure as code scripts. *Inf. Softw. Technol.* 112, 11 (2019). DOI : https://doi.org/10.1016/j.infsof.2019.04.013

[42] Eric Rescorla. 2000. Http over tls. (2000). http://dret.net/biblio/reference/rfc2818.

[43] Johnny Saldana. 2015. *The Coding Manual for Qualitative Researchers*. Sage.

[44] J. H. Saltzer and M. D. Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (September 1975), 1278–1308. DOI : https://doi.org/10.1109/PROC.1975.9939

[45] Stefan Schmidt. 2009. Shall we really do it again? The powerful concept of replication is neglected in the social sciences. *Rev. Gen. Psychol.* 13, 2 (2009), 90–100.

[46] Julian Schwarz. 2017. Code Smell Detection in Infrastructure as Code. Retrieved July 2, 2019 from https://www.swc.rwth-aachen.de/thesis/code-smell-detection-infrastructure-code/.

[47] J. Schwarz, A. Steffens, and H. Lichter. 2018. Code smells in infrastructure as code. In *Proceedings of the 2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC'18)*. 220–228. DOI : https://doi.org/10.1109/QUATIC.2018.00040

[48] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does your configuration code smell? In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR'16)*. ACM, New York, NY, 189–200. DOI : https://doi.org/10.1145/2901739.2901761

[49] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR'05)*. ACM, New York, NY, 1–5. DOI : https://doi.org/10.1145/1082983.1083147

[50] Margaret-Anne Storey, Jody Ryall, R. Ian Bull, Del Myers, and Janice Singer. 2008. TODO or to bug: Exploring how task annotations play a role in the work practices of software developers. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*. ACM, New York, NY, 251–260. DOI : https://doi.org/10.1145/1368088.1368123

[51] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. Icomment: Bugs or bad comments? In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*. ACM, New York, NY, 145–158. DOI : https://doi.org/10.1145/1294261.1294276

[52] Eduard van der Bent, Jurriaan Hage, Joost Visser, and Georgios Gousios. 2018. How good is your puppet? An empirically defined and validated quality model for puppet. In *Proceedings of the 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER'18)*. 164–174. DOI : https://doi.org/10.1109/SANER.2018.8330206

[53] Xiaoyun Wang and Hongbo Yu. 2005. How to break MD5 and other hash functions. In *Proceedings of the 24th Annual International Conference on Theory and Applications of Cryptographic Techniques (EUROCRYPT'05)*. Springer-Verlag, Berlin, 19–35. DOI : https://doi.org/10.1007/11426639_2

[54] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. 2012. *Experimentation in Software Engineering*. Springer, Berlin.

[55] Yevgeniy Brikman. 2016. Why we use Terraform and not Chef, Puppet, Ansible, SaltStack, or CloudFormation. Retrieved April 24, 2020 from https://blog.gruntwork.io/why-we-use-terraform-and-not-chef-puppet-ansible-saltstack-or-cloudformation-7989dad2865c.

[56] Tatu Ylonen and Chris Lonvick. 2006. The secure shell (SSH) protocol architecture. (2006). https://tools.ietf.org/html/rfc4251.