Security Smells in Infrastructure as Code Scripts

Akond Rahman, Md. Rayhanur Rahman, Chris Parnin *Member, IEEE*, and Laurie Williams *Fellow, IEEE*

Abstract—**Context:** Security smells are coding patterns in source code that are indicative of security weaknesses. As infrastructure as code (IaC) 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 paper is to help practitioners avoid insecure coding practices while developing infrastructure as code (IaC) scripts through an empirical study of security smells in IaC scripts.* **Methodology:** We apply qualitative analysis with 3,339 IaC scripts to identify security smells for IaC scripts written in three languages: Ansible, Chef, and Puppet. We construct a static analysis tool called **S**ecurity **L**inter for **I**nfrastructure as **C**ode scripts (SLIC) to automatically identify security smells in 61,097 scripts collected from 1,093 open source software repositories. We also submit bug reports for 1,500 randomly-selected smell occurrences identified from the 61,097 scripts. **Results:** We identify nine security smells for IaC scripts. By applying SLIC on 61,097 IaC scripts we identify 64,356 occurrences of security smells that included 9,092 hard-coded passwords. We observe agreement for 130 of the responded 187 bug reports, which suggests the relevance of security smells for IaC scripts amongst practitioners. **Conclusion:** We observe security smells to be prevalent in IaC scripts. We recommend practitioners to rigorously inspect the presence of the identified security smells in IaC scripts using (i) code review, and (ii) static analysis tools.

Index Terms—ansible, chef, devops, devsecops, empirical study, infrastructure as code, puppet, security, smell, static analysis

1 Introduction

Infrastructure as code (IaC) scripts help practitioners to provision and configure their development environment and servers at scale [1]. IaC scripts are also known as configuration scripts [1] [2] or configuration as code scripts [1] [3]. Commercial IaC tool vendors, such as Ansible [1], Chef [2] and Puppet [4], provide programming constructs and libraries so that programmers can specify configuration and dependency information as scripts.

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 (NASA) to reduce its multi-day patching process to 45 minutes [5]. 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 [6].

However, IaC scripts can be susceptible to security weakness. 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 [3]. In the code snippet, we observe the 'gpgcheck' parameter is assigned 'no', indicating while downloading the 'nginx' package, the 'yum' package manager will not check the contents of the downloaded package [4]. Not checking the content of a downloaded package is related to a security weakness called 'Download of Code Without Integrity Check (CWE-494) [5]'. 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*".

As another example let us consider Figure 2. In this figure, we present an actual Puppet code snippet extracted from the 'aeolus-configure' open source software (OSS) repository [6]. We observe a hard-coded password using the 'password' attribute. A hard-coded string 'v23zj59an' is assigned as password for user 'aeolus'. Hard-coded passwords in software artifacts is considered as a software security weakness ('CWE-798: Use of Hard-coded Credentials') by Common Weakness Enumerator (CWE) [7]. According to CWE [7], "*If hard-coded passwords are used, it is almost certain that malicious users will gain access to the account in question*".

The above-mentioned examples suggest that Ansible and Puppet scripts are susceptible to security smells. Security smells are recurring coding patterns that are indicative of security weakness [8]. A security smell may not lead to a security breach, but deserves rigorous review [8]. Existence and persistence of these smells in IaC scripts leave the possibility of another programmer using these smelly scripts, potentially propagating use of insecure coding practices, as well as providing attackers opportunities to attack the provisioned system. We hypothesize through systematic empirical analysis, we can identify security smells and the prevalence of the identified security smells for IaC scripts.

The goal of this paper is to help practitioners avoid insecure coding practices while developing infrastructure as code (IaC) scripts through an empirical study of security smells in IaC scripts.

We answer the following research questions:

- **RQ1**: What security smells occur in infrastructure as code scripts?

Received: date / Accepted: date.

1. https://www.ansible.com/
2. https://www.chef.io/chef/
3. https://git.openstack.org/cgit/openstack/openstack-ansible-ops/
4. https://docs.ansible.com/ansible/2.3/yum_repository_module.html
5. https://cwe.mitre.org/data/definitions/494.html
6. https://github.com/aeolusproject/aeolus-configure

```
- 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.

```
postgres::user{"aeolus":
               password => "v23zj59an",
               roles => "CREATEDB",
               require => Service["postgresql"] }
```

Hard-coded password

Fig. 2. An example IaC script with hard-coded password.

- **RQ2**: How frequently do security smells occur for infrastructure as code scripts?
- **RQ3**: How do practitioners perceive the identified security smell occurrences for infrastructure as code scripts?

We build on prior research [8] related to security smells for IaC scripts, and investigate what security smells for three languages used to implement the practice of IaC namely, Ansible, Chef, and Puppet. We apply qualitative analysis [9] on 1,101 Ansible scripts, 855 Chef, and 1,721 Puppet scripts to determine security smells. Next, we construct a static analysis tool called **S**ecurity **L**inter for **I**nfrastructure as **C**ode scripts (**SLIC**) [8] to automatically identify the occurrence of these security smells in 14,253 Ansible, 36,070 Chef, and 10,774 Puppet scripts collected by respectively, mining 365, 449, and 280 OSS repositories. We calculate smell density for each type of security smell in the collected IaC scripts. We submit bug reports for 1500 randomly-selected smell occurrences for Ansible, Chef, and Puppet to assess the relevance of the identified security smells.

Contributions: Compared to prior research [8] in which we reported findings specific to Puppet, we make the following additional contributions:

- A list of security smells with definitions for Ansible and Chef scripts;
- An evaluation of how frequently security smells occur in Ansible and Chef scripts;
- An evaluation of how practitioners perceive the identified security smells for Ansible and Chef scripts; and
- An empirically-validated tool (SLIC) that automatically detects occurrences of security smells for Ansible, and Chef scripts.

This paper combines answers to the research questions for all three languages: Ansible, Chef, and Puppet.

We organize the rest of the paper as following: we provide background information with related work discussion in Section 2. We describe the methodology and the definitions of seven security smells in Section 3. We describe the methodology to construct and evaluate SLIC in Section 4. In Section 5.1, we describe the methodology for our empirical study. We report our findings in Section 6, followed by a discussion in Section 7. We describe limitations in Section 8, and conclude our paper 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, Chef, and Puppet scripts, along with CWE, as we use CWE to validate our qualitative process described in Section 3.1.

2.2 Ansible, Chef, and Puppet Scripts

IaC is the practice of automatically defining and managing network and system configurations and infrastructure through source code [1]. Companies widely use commercial tools such as Puppet, to implement the practice of IaC [1] [10] [11]. For example, using IaC scripts application deployment time for Borsa Istanbul, Turkey's stock exchange, reduced from ∼10 days to an hour [12]. With IaC scripts Ambit Energy increased their deployment frequency by a factor of 1,200 [13].

A 2019 survey with 786 practitioners reported Ansible as the most popular language to implement IaC, followed by Chef and Puppet [7] [8]. As usage of Ansible, Chef, and Puppet is getting increasingly popular amongst practitioners, identification of security smells could have relevance to practitioners in mitigating insecure coding practices in IaC.

We provide a brief background on Ansible, Chef, and Puppet scripts, which is relevant to conduct our empirical study. Both, Ansible and Chef provides 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 3, 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, which creates an empty file.

In the case of Chef, configurations as specified using 'recipes', which are domain-specific Ruby scripts. Dedicated libraries are also available to maintain certain configurations. As shown in Figure 4, 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.

Typical entities of Puppet include manifests [4]. Manifests are written as scripts that use a .pp extension. In a single manifest script, configuration values can be specified using variables and attributes. For better understanding, we provide a sample Puppet script with annotations in Figure 5. For attributes configuration values are specified using the '=>' sign. As shown in Figure 5, an empty file '/tmp/sample.txt' is created using the 'file' resource provided by Puppet [4]. The attributes of the resource file such as, path, owner, group and content can also be specified.

7. https://info.flexerasoftware.com/SLO-WP-State-of-the-Cloud-2019
8. https://www.techrepublic.com/article/ansible-overtakes-chef-and-puppet-as-the-top-cloud-configuration-management-tool/

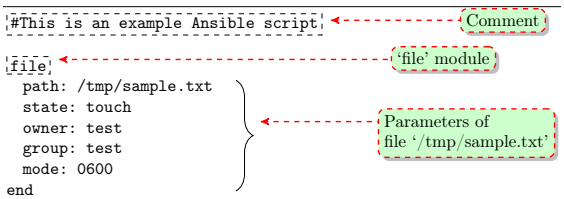

Fig. 3. Annotation of an example Ansible script.

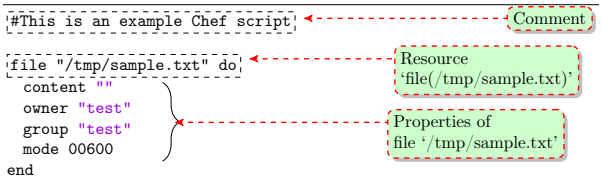

Fig. 4. Annotation of an example Chef script.

2.3 Common Weakness Enumeration (CWE)

The Common Weakness Enumeration (CWE) is a community-driven database for software security weaknesses and vulnerabilities [7]. 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 [7]. 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 [7].

2.4 Related Work

For IaC scripts, we observe a lack of studies that investigate coding practices with security consequences. For example, Sharma et al. [2], Schwarz [14], and Bent et al. [15], in separate studies investigated code maintainability aspects of Chef and Puppet scripts Jiang and Adams [10] investigated the co-evolution of IaC scripts and other software artifacts, such as build files and source code. Rahman and Williams [16] characterized defective IaC scripts using text mining and created prediction models using text feature metrics. Rahman et al. [17] surveyed practitioners to investigate which factors influence usage of IaC tools. Rahman et al. [18] 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 [19] identified 10 code properties in IaC scripts that show correlation with defective IaC scripts. Hanappi et al. [20] investigated how convergence of IaC scripts can be automatically tested, and proposed an automated model-based test framework. In this paper we build upon the research conducted by Rahman et al. [8]'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. Unlike Puppet, Ansible and Chef are both procedural, and the derived list of security smells for Ansible and Chef may be different to that of Puppet. We extend Rahman et al. [8]'s research for Ansible and Chef scripts.

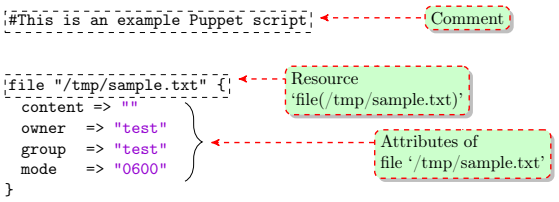

Fig. 5. Annotation of an example Puppet script.

3 Security Smells

A code smell is a recurrent coding pattern that is indicative of potential maintenance problems [21]. A code smell may not always have bad consequences, but still deserves attention, as a code smell may be an indicator of a problem [21]. Our paper focuses on identifying security smells. Security smells are recurring coding patterns that are indicative of security weakness, and requires further inspection [8].

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 infrastructure as code scripts?

Data Collection: We collect a set of Ansible, Chef, and Puppet 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[9]. We collect 855 Chef scripts from 11 repositories maintained by Openstack. We collect 1,383 Puppet scripts that we use to determine the security smells from 74 OSS repositories collected from three organizations: Mozilla, Openstack and Wikimedia Commons.

Qualitative Analysis: We first apply a qualitative analysis technique called descriptive coding [22] on the collected scripts. 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 authors security expertise. Figure 6 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 6. 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 [7]. 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 [7]. A mapping

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

between a derived security smell and a security weakness reported by CWE can validate our qualitative process. For the example presented in Figure 6, 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' [7].

The first and second author, who are both PhD students, individually conduct descriptive coding process for Ansible and Chef. The first author applied qualitative analysis to determine security smells for Puppet. Upon completion of the descriptive coding process, we record the agreements and disagreements for the identified security smells. We also calculate Cohen's Kappa [23].

For Ansible, the first and the second author respectively identified four and six security smells. Upon discussion, the first and second author agreed upon one additional security smell. For Chef, the first and the second author respectively identified seven and nine security smells. Upon discussion, the first and second author agreed upon including one security smell and excluding one security smell. The Cohen's Kappa is respectively, 0.6 and 0.5 for Ansible and Chef scripts between the first and second author of the paper. For Puppet, the first author derived the seven smells, and the derivation process is subject to the first author's judgment. We mitigate this limitation by recruiting two independent raters who are not authors of the paper. These two raters, with background in software security, independently evaluated if the identified smells are related to the associated CWEs. Both raters mapped each of the seven security smells to the same CWEs. We observe a Cohen's Kappa score of 1.0 between raters.

3.2 Answer to RQ1: What security smells occur in infrastructure as code 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 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 TLS, and use of weak cryptography algorithm. For Puppet scripts, we identify seven security smells: admin by default, empty password, hard-coded secret, suspicious comment, unrestricted IP address, use of HTTP Without 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 TLS. Examples of each security smell for Ansible, Chef, and Puppet are respectively, presented in Figure 7, 8, and 9. Below, we list the names of the smells alphabetically, where each smell name is followed by the applicable language: Ansible (), Chef (), and Puppet ().

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 [24], 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 8, 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' [7].

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 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 [25]. 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' [7].

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' [7].

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' [7].

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' [7].

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

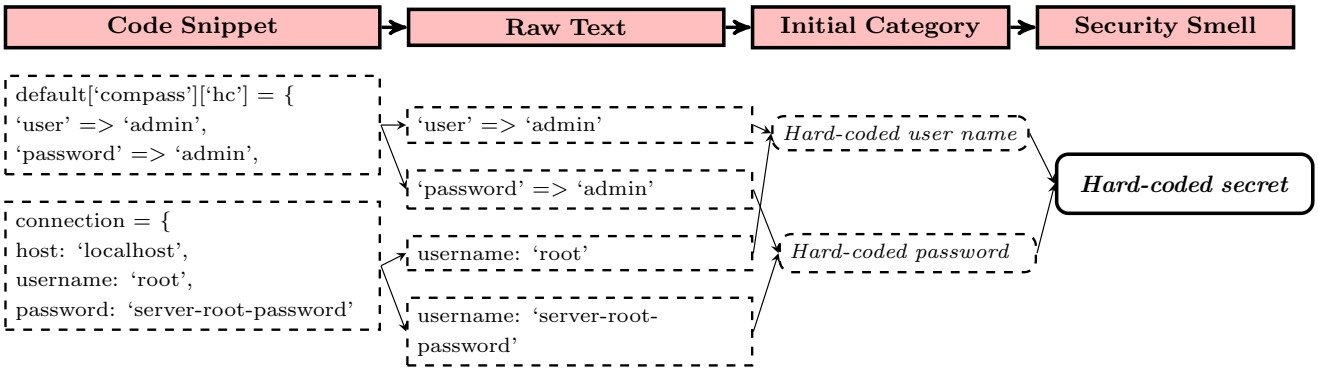

Fig. 6. An example to demonstrate the process of determining security smells using open coding.

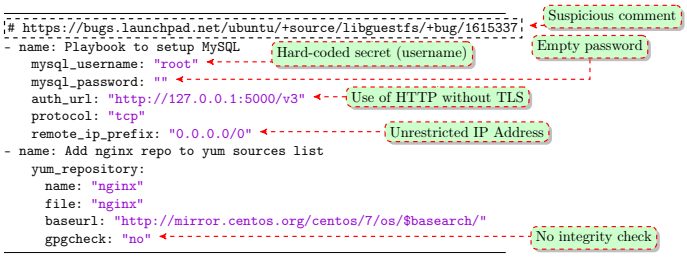

Fig. 7. An annotated Ansible script with six security smells. The name of each security smell is highlighted on the right.

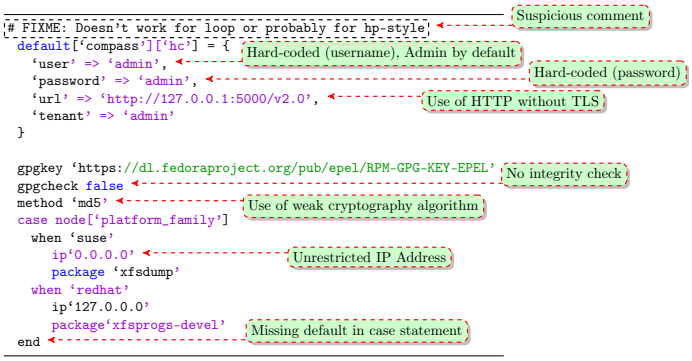

Fig. 8. An annotated Chef script with eight security smells. The name of each security smell is highlighted on the right.

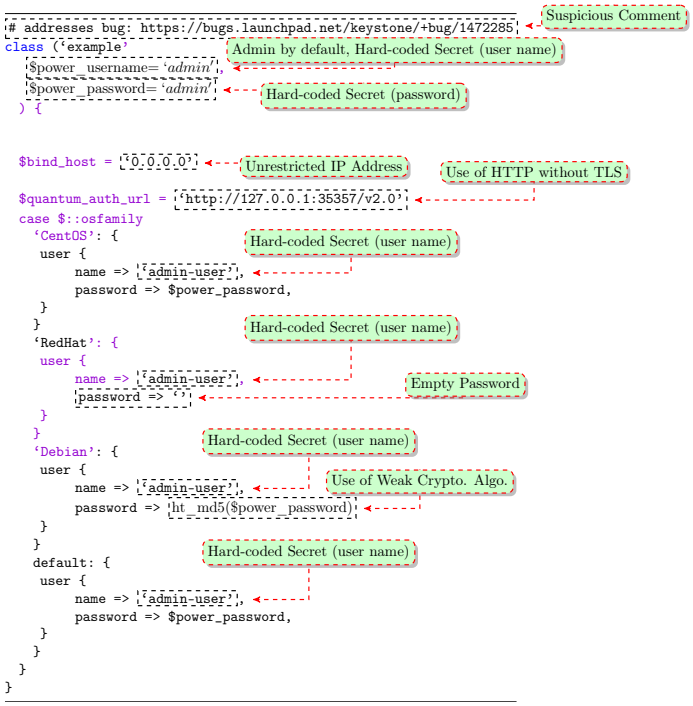

Fig. 9. An annotated Puppet script with seven security smells. The name of each security smell is highlighted on the right.

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 [26]. However, these keywords make a comment 'suspicious'. The smell is related to 'CWE-546: Suspicious Comment' [7].

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 [27]. 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 [8]. This smell is related to improper access control as stated in the weakness 'CWE-284: Improper Access Control' [7].

Use of HTTP without TLS () : This smell is the recurring pattern of using HTTP without the Transport Layer Security (TLS). Such use makes the communication between two entities less secure, as without TLS, use of

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

HTTP is susceptible to man-in-the-middle attacks [28]. For example, as shown in Figure 7, the authentication URL uses HTTP without 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 TLS' may not lead to a security attack. This security smell is related to 'CWE-319: Cleartext Transmission of Sensitive Information' [7].

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 [29]. MD5 is susceptible to collision attacks [30] and modular differential attacks [31]. Similar 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' [7].

4 Security Linter for Infrastructure as Code Scripts (SLIC)

We first describe how we constructed SLIC, then we describe how we evaluated SLIC's smell detection accuracy.

4.1 Description of SLIC

SLIC is a static analysis tool for detecting security smells in IaC scripts. SLIC has two components:

Parser: The Parser parses an Ansible, Chef, or Puppet 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 10a and 11a respectively provides a sample script in Ansible and Chef that is fed into SLIC. The output of Parser is expressed as a vector, as shown in Figures 10b and 11b. For example in Figure 11b, 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.

In the case of Chef and Puppet, each token is marked with its name, token type, and any associated configuration value, except for comments. Only token type and configuration value are marked for comments. For example, Figure 12a provides a sample script that is fed into SLIC. The output of Parser is is expressed as a vector, as shown in Figure 12b. For example, the comment in line#1, is expressed as the vector '<COMMENT, 'This is an example Puppet script'>'. Parser provides a vector representation of all code snippets in a script.

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

a

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'>

b

Fig. 10. Output of the 'Parser' component in SLIC. Figure 10a presents an example Ansible script fed to Parser. Figure 10b presents 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
```

a

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

b

Fig. 11. Output of the 'Parser' component in SLIC. Figure 11a presents an example Chef script fed to Parser. Figure 11b presents the output of Parser for the example Chef script.

Rule Engine: We take motivation from prior work [32] and 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 [32]; and (ii) rules can be applicable for IaC tools irrespective of their 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 abstract patterns from the smell-related code snippets and constitute a rule from the generated patterns. We use Table 1 to demonstrate our approach. The 'Code Snippet' column presents a list of code snippets related to 'Use of HTTP without 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 TLS'. The vectors can be abstracted to construct the following rule: '$(isVariable(x) \vee isProperty(x)) \wedge isHTTP(x)$'. 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

13. https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html

```
1 #This is an example Puppet script
2 $token = `XXXYYYZZZ'
3 $os_name = `Windows'
4 auth_protocol => `http'
5 $vcenter_password = `password'
```

a

Line#	Output of Parser
1	<COMMENT, 'This is an example Puppet script'>
2	<VARIABLE, 'token', 'XXXYYYZZZ'>
3	<VARIABLE, 'os_name', 'Windows'>
4	<ATTRIBUTE, 'auth_protocol', 'http'>
5	<VARIABLE, 'vcenter_password', 'password'>

b

Fig. 12. Output of the 'Parser' component in SLIC. Figure 12a presents an example Puppet script fed to Parser. Figure 12b presents the output of Parser for the example Puppet script.

TABLE 1
An Example of Using Code Snippets To Determine Rule for 'Use of HTTP Without 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' >

which is related to specifying a URL that uses HTTP without TLS support, then the script contains the security smell 'Use of HTTP without 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 SLIC to identify security smells for one or multiple Ansible, Chef, and Puppet scripts. The programmer specifies a directory where script(s) reside. Upon completion of analysis, SLIC generates a comma separated value (CSV) file where the count of security smell for each script is reported. We implement SLIC using API methods provided by PyYAML [14] for Ansible, Foocritic [15] for Chef, and the puppet-lint API [16] for Puppet.

Rules to Detect Security Smells: For Ansible, Chef, and Puppet we present the rules needed for the 'Rule Engine' of SLIC respectively in Tables 2, 3, and 4. The string patterns needed to support the rules in Tables 2, 3, and 4 are listed in Table 5. 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 [33] [34], where we check if the value satisfies the necessary condition. Table 5 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[#nt]$*$[0-9]+' is satisfied.

For Ansible, Chef, and Puppet scripts the rule engine takes output from the Parser, and checks if any of the rules listed in Tables 2, 3, and 4 respectively, for Ansible, Chef, and Puppet. In Tables 2, 3, and 4 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 2. Similarly, in the case of Chef scripts, we use the output of Parser to variables ($isVariable(x)$), properties ($isProperty(x)$), attributes ($isAttribute(x)$), and case statements ($isCaseStmt(x)$). In the case of Puppet scripts, to detect whether or not a token type is a variable ($isVariable(x)$), an attribute ($isAttribute(x)$), a function ($isFunction(x)$), 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.

14. https://pyyaml.org/
15. http://www.foodcritic.io/
16. http://puppet-lint.com/

4.2 Evaluation of SLIC

Any static analysis tool is subject to evaluation. We use raters to construct the oracle dataset to mitigate author bias in SLICs evaluation, similar to Chen et al. [36].

Oracle Dataset for Ansible and Chef: For Ansible and Chef, we construct two oracle datasets using closed coding [22], where a rater identifies a pre-determined pattern. For the Ansible oracle dataset, 96 scripts are manually checked for security smells by at least two raters. For the Chef oracle dataset, 76 scripts are manually checked for security smells by at least two raters. The raters apply their knowledge related to IaC scripts and software security to determine if a certain smell appears for a script.

We made the smell identification task available to the raters using a website [17]. In each task, a rater determines which of the security smells identified in Section 3.1 occur in a script. To avoid bias, we did not include any raters who were involved in deriving smells or constructing SLIC. 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 the year of 2019 at North Carolina State University. Of the 60 students in the class, 32 students agreed to participate. We assigned 96 Ansible and 76 Chef scripts scripts to the 32 students to ensure each script is reviewed by at least two students, where each student does not have to rate more than 15 scripts. We used balanced block design [37] to assign 96 Ansible and 76 Chef scripts from our collection of 1,101 Ansible and 855 Chef scripts. For Ansible, we observe agreements on the rating for 54 of 96 scripts (56.2%), with a Cohen's Kappa of 0.3. 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 [38], the reported agreement is 'fair' and 'moderate' respectively, for Ansible and Chef. In the case of disagreements between raters, the first author resolved the disagreements.

Upon completion of the oracle dataset, we evaluate the accuracy of SLIC 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 SLIC. Recall refers to the fraction of correctly identified smells that have been retrieved by SLIC over the total amount of security smells.

Oracle Dataset for Puppet: We construct the oracle dataset by applying closed coding [22], where a rater iden-

17. http://13.59.115.46/website/start.php

TABLE 2
Rules to Detect Security Smells for Ansible Scripts

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

TABLE 3
Rules to Detect Security Smells for Chef Scripts

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

TABLE 4
Rules to Detect Security Smells for Puppet Scripts

Smell Name	Rule
Admin by default	($isParameter(x)$) $\wedge$ ($isAdmin(x.name) \wedge isUser(x.name)$)
Empty password	($isAttribute(x) \vee isVariable(x)$) $\wedge$ (($length(x.value) == 0 \wedge isPassword(x.name)$)
Hard-coded secret	($isAttribute(x) \vee isVariable(x)$) $\wedge$ ($isUser(x.name) \vee isPassword(x.name) \vee isPvtKey(x.name)$) $\wedge$ ($length(x.value)>0$)
Suspicious comment	($isComment(x)$) $\wedge$ ($hasWrongWord(x) \vee hasBugInfo(x)$)
Unrestricted IP address	(($isVariable(x) \vee isAttribute(x)$) $\wedge$ ($isInvalidBind(x.value)$)
Use of HTTP without TLS	($isAttribute(x) \vee isVariable(x)$) $\wedge$ ($isHTTP(x.value)$)
Use of weak crypto. algo.	($isFunction(x) \wedge usesWeakAlgo(x.name)$)

TABLE 5
String Patterns Used for Functions in Rules

Function	String Pattern
$hasBugInfo()$ [35]	'bug[#*nt*]*[0-9]+','show_bug\.cgi?id=[0-9]+'
$hasWrongWord()$ [7]	'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]+'
$isUser()$	'user'
$usesWeakAlgo()$	'md5', 'sha1'

tifies a pre-determined pattern. In the oracle dataset, 140 Puppet scripts are manually checked for security smells by at least two raters. The raters apply their knowledge related to Puppet scripts and security, and determine if a certain smell appears for a script. To avoid bias, we did not include any raters who were involved in deriving smells or constructing SLIC, similar to Ansible and Chef.

We made the smell identification task available to the raters using a website. In each task, a rater determines which of the security smells identified in Section 3.1 occur in a script. 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 the year of 2018 at North Carolina State University. We obtained IRB approval for the student participation. Of the 58 students in the class, 28 students agreed to participate. We assigned 140 scripts to the 28 students to ensure each script is reviewed by at least two students, where each student does not have to rate more than 10 scripts. We used balanced block design [37] to assign 140 scripts from our collection of 1,383 scripts. We observe agreements on the rating for 79 of 140 scripts (56.4%), with a Cohen's Kappa of 0.3. According to Landis and Koch's interpretation [38], the reported agreement is 'fair'. In the case of disagreements between raters for 61 scripts, the first author resolved the disagreements.

Performance of SLIC for Ansible and Chef Oracle Dataset: We report the detection accuracy of SLIC with respect to precision and recall For Ansible in Table 6. As

shown in the 'No smell' row, we identify 77 Ansible scripts with no security smells. The count of occurrences for each security smell along with SLIC's precision and recall for the Ansible and Chef oracle dataset are provided in Table 6 and 7.

TABLE 6
SLIC's Accuracy for the Ansible Oracle Dataset

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 TLS	14	1.0	1.0
No smell	77	1.0	1.0
Average		1.0	1.0

TABLE 7
SLIC's Accuracy for the Chef Oracle Dataset

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 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

Performance of SLIC for Puppet Oracle Dataset: We report the detection accuracy of SLIC with respect to precision and recall in Table 8 for Puppet scripts.

Average precision and recall of SLIC is $>= 0.9$ for all indeitifed security smells and the category 'no smell'.

TABLE 8
SLIC's Accuracy for the Puppet Oracle Dataset

Smell Name	Occurr.	Precision	Recall
Admin by default	1	1.0	1.0
Empty password	2	1.0	1.0
Hard-coded secret	24	1.0	0.9
Suspicious comment	17	1.0	1.0
Unrestricted IP address	4	1.0	1.0
Use of HTTP without TLS	9	1.0	1.0
Use of weak crypto. algo.	1	1.0	1.0
No smell	113	0.9	1.0
Average		0.9	0.9

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

5 Empirical Study

Using SLIC, we conduct an empirical study to quantify the prevalence of security smells in IaC scripts in three languages: Ansible, Chef, and Puppet.

5.1 Research Questions

We investigate the following research questions:

- RQ2: How frequently do security smells occur for infrastructure as code scripts?
- RQ3: How do practitioners perceive the identified security smell occurrences for infrastructure as code scripts?

5.2 Datasets

We conduct our empirical study with six datasets: two datasets each for Ansible, Chef, and Puppet scripts. We construct three datasets from repositories maintained by Openstack. The other three 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 tend to host their popular OSS projects on GitHub [40] [41].

As advocated by prior research [42], OSS repositories need to be curated. We apply the following criteria to curate the collected repositories:

- **Criteria-1**: At least 11% of the files belonging to the repository must be IaC scripts. Jiang and Adams [10] reported for OSS repositories, which are used in production, IaC scripts co-exist with other types of files, such as Makefiles. They observed a median of 11% of the files to be IaC scripts. By using a cutoff of 11%, we assume to collect repositories that contain sufficient amount of IaC scripts for analysis.
- **Criteria-2**: The repository is not a clone.
- **Criteria-3**: The repository must have at least two commits per month. Munaiah et al. [42] used the threshold of at least two commits per month to determine which repositories have enough software development activity. We use this threshold to filter repositories with little activity.
- **Criteria-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 [43] [41].

We answer RQ2 using 14,253 Ansible, 36,070 Chef, and 10,774 Puppet scripts, respectively, collected from 365, 449, and 280 repositories. Table 9 summarizes how many repositories are filtered out using our criteria. We clone the master branches of these repositories. Summary attributes of the collected repositories are available in Table 10.

5.3 Analysis

5.3.1 RQ2: How frequently do security smells occur for infrastructure as code scripts?

First, we apply SLIC to determine the security smell occurrences for each script. Second, we calculate two metrics described below:

- ***Smell Density***: Similar to prior research that have used defect density [44] and vulnerability density [45], we use smell density to measure the frequency of a security smell x, for every 1000 lines of code (LOC). We measure smell density using Equation 1.

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

- ***Proportion of Scripts (Script%)***: Similar to prior work in defect analysis [46], 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.

TABLE 9
OSS Repositories Satisfying Criteria (Sect. 5.2)

	Ansible		Chef		Puppet	
	GH	OST	GH	OST	GH	OST
Initial Repo Count	3,405,303	1,253	3,405,303	1,253	3,405,303	1,253
Criteria-1 (11% IaC Scripts)	13,768	16	5,472	15	6,088	67
Criteria-2 (Not a Clone)	10,017	16	3,567	11	4,040	61
Criteria-3 (Commits/Month $\geq$ 2)	10,016	16	3,565	11	2,711	61
Criteria-4 (Devs $\geq$ 10)	349	16	438	10	219	61
Final Repo Count	349	16	438	10	219	61

TABLE 10
Summary Attributes of the Datasets

	Ansible		Chef		Puppet	
Attribute	**GH**	**OST**	**GH**	**OST**	**GH**	**OST**
Repository Count	349	16	438	11	219	61
Total File Count	498,752	4,487	126,958	2,742	72,817	12,681
Total Script Count	13,152	1,101	35,132	938	8,010	2,764
Tot. LOC (IaC Scripts)	602,982	52,239	1,981,203	63,339	424,184	214,541

The two metrics characterize the frequency of security smells differently. The smell density metric is more granular, and focuses on the content of a script as measured by how many smells occur for every 1000 LOC. The proportion of scripts metric is less granular and focuses on the existence of at least one of the seven security smells for all scripts.

5.3.2 RQ3: How do practitioners perceive the identified security smell occurrences for infrastructure as code 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, Chef, and Puppet 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 SLIC 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 the two research questions as following:

6.1 RQ2: How frequently do security smells occur for infrastructure as code 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. For Puppet, in GitHub and Openstack datasets we observe proportion of scripts to be respectively, 29.3% and 32.9%. Hard-coded secret is the most prevalent security smell with respect to occurrences, smell density, and proportion of scripts contain hard-coded secrets. A complete breakdown of findings related to RQ2 for Ansible, Chef, and Puppet is presented in Tables 11, 12, and 13 for our datasets.

Occurrences: The occurrences of the security smells are presented in the 'Occurrences' column of Table 11 for all six 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 Puppet we observe 17,756 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. For Puppet, we identify 14,444 occurrences of hard-coded secrets, of which 68.6%, 22.9%, and 8.5% are respectively, hard-coded keys, user names, and passwords.

Exposing hard-coded secrets, such as hard-coded keys, is not uncommon in GitHub: Meli et al. [47] identified 201,642 instances of private keys, which included commonly-used API keys. Meli et al. [47] reported 85,311 of the identified 201,642 instances of private keys to be Google API keys.

Smell Density: In Table 12, we report the smell density for all three languages. The 'Combined' row presents the smell density for each dataset when all seven security smell occurrences are considered. For all six datasets, we observe the dominant security smell to be 'Hard-coded secret'.

Proportion of Scripts (Script%): In Table 13, we report the proportion of scripts (Script %) values for each of the six datasets. The 'Combined' row represents the proportion of scripts in which at least one of the identified smells appear.

TABLE 11
Smell Occurrences for Ansible, Chef, and Puppet scripts

	Ansible		Chef		Puppet	
Smell Name	**GH**	**OST**	**GH**	**OST**	**GH**	**OST**
Admin by default	N/A	N/A	301	61	52	35
Empty password	298	3	N/A	N/A	136	21
Hard-coded secret	14,409	722	14,160	1,203	10,892	3,552
Missing default in switch	N/A	N/A	953	68	N/A	N/A
No integrity check	194	14	2,249	132	N/A	N/A
Suspicious comment	1,421	138	3,029	161	758	305
Unrestricted IP address	129	7	591	19	188	114
Use of HTTP without TLS	934	84	4,898	326	1,018	460
Use of weak crypto algo.	N/A	N/A	94	2	177	48
Combined	17,385	968	26,275	1,972	13,221	4,535

TABLE 12
Smell Density for Ansible, Chef, and Puppet scripts

	Ansible		Chef		Puppet	
Smell Name	**GH**	**OST**	**GH**	**OST**	**GH**	**OST**
Admin by default	N/A	N/A	0.1	0.9	0.1	0.1
Empty password	0.49	0.06	N/A	N/A	0.3	0.1
Hard-coded secret	23.9	13.8	7.1	19.0	25.6	16.5
Missing default in switch	N/A	N/A	0.5	1.0	N/A	N/A
No integrity check	0.3	0.2	1.1	2.1	N/A	N/A
Suspicious comment	2.3	2.6	1.5	2.5	1.7	1.4
Unrestricted IP address	0.2	0.1	0.3	0.3	0.4	0.5
Use of HTTP without TLS	1.5	1.6	2.4	5.1	2.4	2.1
Use of weak crypto algo.	N/A	N/A	0.05	0.03	0.4	0.1
Combined	28.8	18.5	13.3	31.5	25.3	29.6

TABLE 13
Proportion of Scripts With At Least One Smell for Ansible, Chef, and Puppet scripts

	Ansible		Chef		Puppet	
Smell Name	**GH**	**OST**	**GH**	**OST**	**GH**	**OST**
Admin by default	N/A	N/A	0.3	2.1	0.6	1.1
Empty password	1.1	0.2	N/A	N/A	1.4	0.5
Hard-coded secret	19.2	22.4	6.8	15.9	21.9	24.8
Missing default in switch	N/A	N/A	2.5	6.5	N/A	N/A
No integrity check	1.1	1.0	3.6	3.8	N/A	N/A
Suspicious comment	6.3	8.0	6.6	9.3	5.9	7.2
Unrestricted IP address	0.5	0.4	1.1	1.0	1.7	2.9
Use of HTTP without TLS	3.7	3.0	4.9	6.9	6.3	8.5
Use of weak crypto algo.	N/A	N/A	0.2	0.1	0.9	0.5
Combined	25.3	29.6	20.5	30.4	29.3	32.9

6.2 RQ3: How do practitioners perceive the identified security smell occurrences for infrastructure as code scripts?

From 7 and 19 repositories, respectively, we obtain 29 and 54 responses for Ansible and Chef scripts. In the case of Ansible scripts, we observe an agreement of 82.7% for 29 smell occurrences. For Chef scripts, we observe an agreement of 55.5% for 54 smell occurrences. The percentage of smells to which practitioners agreed to be fixed for Ansible and Chef is respectively, presented in Figure 13 and 14. In the y-axis each smell name is followed by the occurrence count. For example, according to Figure 13, for 4 occurrences of 'Use of HTTP without TLS' (HTTP.USG) , we observe 100% agreement for Ansible scripts. In the case of Puppet, from 21 repositories, we obtain 104 responses for the submitted 500 bug reports in terms of acknowledgement or changing the source code. We observe an agreement of 63.4% for 104 smell occurrences. The percentage of smells to which practitioners agreed to be fixed is presented in Figure 15. In the y-axis each smell name is followed by the occurrence count. For example, according to Figure 15, for 9 occurrences of 'Use of weak cryptography algorithms'(WEAK_CRYP), we observe 77.7% agreement. We observe 75.0% or more agreement for one smell: 'Use of weak cryptography algorithms'.

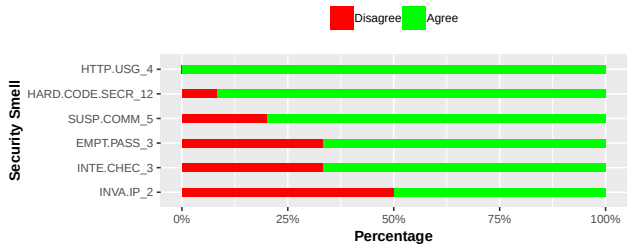

Fig. 13. Feedback for 29 smell occurrences for Ansible. Practitioners agreed with 82.7% of the selected smell occurrences.

Reasons for Practitioner Agreements: In their response, practitioners provided reasoning on why these smells ap-

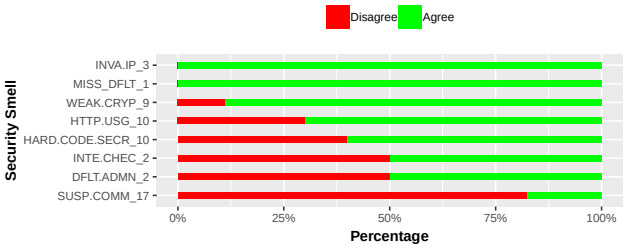

Fig. 14. Feedback for the 54 smell occurrences for Chef. Practitioners agreed with 55.5% of the selected smell occurrences.

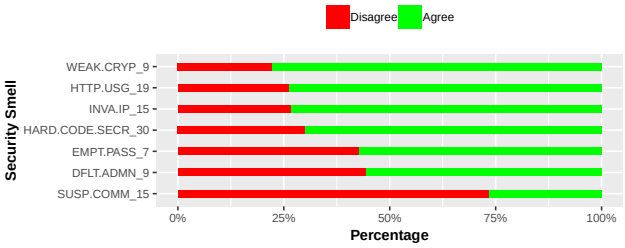

Fig. 15. Feedback for the 104 smell occurrences. Practitioners agreed with 63.4% of the selected smell occurrences.

peared. For one occurrence of 'HTTP without TLS' in a Chef script, one practitioner suggested availability of a HTTPS endpoint saying: "*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*". For an occurrence of hard-coded secret in an Ansible script one practitioner agreed stating possible solutions: "*I agree that it [hard-coded secret] could be in an Ansible vault or something dedicated to secret storage.*". Upon acceptance of the smell occurrences, practitioners also suggested how these smells can be mitigated. For example, for an occurrence of 'Unrestricted IP Address' in a Puppet script, one practitioner stated:"*I would accept a pull request to do a default of 127.0.0.1*".

Reasons for Practitioner Disagreements: We observe practitioners to value development context when disagreeing with security smell occurrences. For example, a hard-coded password may not have security implications for practitioners if the hard-coded password is used for testing purposes. One practitioner disagreed stating "*the code in question is an integration test. The username and password is not used anywhere else so this should be no issue.*". These anecdotal evidence suggests that while developing IaC scripts practitioners may only be considering their own development context, and not realizing how another practitioner may perceive use of these security smells as an acceptable practice. For one occurrence of 'HTTP Without TLS' in a Puppet script one practitioner disagreed stating "*It's using http on localhost, what's the risk?*".

The above-mentioned statements from disagreeing practitioners also suggest lack of awareness, e.g. if a developer comes across the script of interest mentioned in the above-mentioned paragraph, the developer may perceive use of hard-coded passwords to be an acceptable practice, potentially propagating the practice of hard-coded secrets. As another example, both local and remote sites that use HTTP can be insecure, as considered by practitioners from Google [18] [19]. Possible explanations for disagreements can also be attributed to perception of practitioners: smells in code have subjective interpretation [48], and programmers do not uniformly agree with all smell occurrences [49], [50]. Furthermore, researchers [51] have observed programmers' bias to perceive their code snippets as secure, even if the code snippets are insecure.

7 Discussion

We suggest strategies on how the identified security smells can be mitigated along with other implications:

7.1 Mitigation Strategies

Admin by default: We advise practitioners to create user accounts that have the minimum possible security privilege and use that account as default. Recommendations from Saltzer and Schroeder [52] may be helpful in this regard.

Empty password: We advocate against storing empty passwords in IaC scripts. Instead, we suggest the use of strong passwords.

Hard-coded secret: We suggest the following measures to mitigate hard-coded secrets:

- use tools such as Ansible/AWS [20] and Vault [21] to store secrets
- scan IaC scripts to search for hard-coded secrets using tools such as CredScan [22] and SLIC.

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 [26]. 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 5, 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.

18. https://security.googleblog.com/2018/02/a-secure-web-is-here-to-stay.html
19. https://developers.google.com/web/fundamentals/security/encrypt-in-transit/why-https
20. https://github.com/ansible/awx
21. https://www.vaultproject.io/
22. https://secdevtools.azurewebsites.net/helpcredscan.html

Use of HTTP without TLS: We advocate companies to adopt the HTTP with TLS by leveraging resources provided by tool vendors, such as MySQL [23] and Apache [24]. We advocate for better documentation and tool support so that programmers do not abandon the process of setting up HTTP with TLS.

Use of weak cryptography algorithms: We advise programmers to use cryptography algorithms recommended by the National Institute of Standards and Technology [53] to mitigate this smell.

One possible strategy to mitigate security smells is to develop concrete guidelines on how to write IaC scripts in a secure manner. When constructing guidelines, the IaC community can take the findings of Acar et al. [54] into account, and include easy to understand, task-specific examples on how to write IaC scripts in a secure manner.

7.2 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 TLS. Security smells identified for Puppet are also applicable for Chef and Ansible, which provides further validation to Rahman et al. [8]. We also identify an additional security smell namely 'No Integrity Check', which was not previously identified by Rahman et al. [8].

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' observation [16]: 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 observation from Rahman and Williams [16] we conjecture that similar to defective scripts, IaC scripts with security smells may also include certain operations that make distinguishes them from scripts with no security smells.

7.3 Future Work

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 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 are needed to fix security smells.

23. https://dev.mysql.com/doc/refman/5.7/en/encrypted-connections.html

24. https://httpd.apache.org/docs/2.4/ssl/ssl_howto.html

8 Threats to Validity

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

Conclusion Validity: The derived security smells and their association with CWEs are subject to the rater judgment. We account for this limitation by using two raters who are experienced in software security. The oracle dataset constructed by the raters is susceptible to subjectivity, as the raters' judgment influences appearance of a certain security smell. We mitigate this limitation by allocating each script for closed coding with two raters.

One of the raters involved in the smell derivation process was also involved in deriving smells for Puppet scripts. The derivation process involved with Puppet may have influenced that rater in deriving security smells for Ansible and Chef. We mitigate this limitation by adding another rater in deriving security smells for Ansible and Chef.

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, 855 Chef, and 1,383 Puppet scripts for security smells. In the future, we aim to investigate if more security smells exist.

The detection accuracy of SLIC depends on the constructed rules that we have provided in Tables 2, 3, and 4. We acknowledge that the constructed rules are susceptible of generating false positives and false negatives.

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.

9 Conclusion

IaC scripts help companies to automatically provision and configure their development environment, deployment environment, and servers. Security smells are recurring coding patterns in IaC scripts that are indicative of security weakness and can potentially lead to security breaches. By applying qualitative analysis we identified six, eight, and seven security smells respectively, for Ansible, Chef, and 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 TLS.

Next, we construct a static analysis tool called SLIC using which we analyzed 61,097 IaC scripts. We identify 64,356 security smells by analyzing 61,097 scripts, which included 9,092 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 IaC 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 paper will facilitate further security-related research in the domain of IaC scripts.

Acknowledgments

This research study was funded by the Science of Security Lablet at the North Carolina State University. We would like

to thank the RealSearch group members for their valuable feedback.

REFERENCES

[1] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, 1st ed. Addison-Wesley Professional, 2010.

[2] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 189–200. [Online]. Available: http://doi.acm.org/10.1145/2901739.2901761

[3] A. Rahman, A. Partho, P. Morrison, and L. Williams, "What questions do programmers ask about configuration as code?" in *Proceedings of the 4th International Workshop on Rapid Continuous Software Engineering*, ser. RCoSE '18. New York, NY, USA: ACM, 2018, pp. 16–22. [Online]. Available: http://doi.acm.org/10.1145/3194760.3194769

[4] P. Labs, "Puppet Documentation," https://docs.puppet.com/, 2018, [Online; accessed 01-July-2019].

[5] Ansible, "Nasa: Increasing cloud efficiency with ansible and ansible tower," Ansible, Tech. Rep., January 2019. [Online]. Available: https://www.ansible.com/hubfs/pdf/Ansible-Case-Study-NASA.pdf?hsLang=en-us

[6] M. Leone, "The economic benefits of puppet enterprise," ESG, Tech. Rep., November 2016. [Online]. Available: https://puppet.com/resources/analyst-report/the-economic-benefits-puppet-enterprise

[7] MITRE, "CWE-Common Weakness Enumeration," https://cwe.mitre.org/index.html, 2018, [Online; accessed 02-July-2019].

[8] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19, 2019, to appear. Pre-print: https://akondrahman.github.io/papers/icse19_slic.pdf.

[9] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln, *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

[10] Y. Jiang and B. Adams, "Co-evolution of infrastructure and source code: An empirical study," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 45–55. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820527

[11] R. Shambaugh, A. Weiss, and A. Guha, "Rehearsal: A configuration verification tool for puppet," *SIGPLAN Not.*, vol. 51, no. 6, pp. 416–430, Jun. 2016. [Online]. Available: http://doi.acm.org/10.1145/2980983.2908083

[12] P. Labs, "Borsa istanbul: Improving efficiency and reducing costs to manage a growing infrastructure," Puppet, Tech. Rep., July 2018. [Online]. Available: https://puppet.com/resources/case-study/borsa-istanbul

[13] Puppet, "Ambit energy's competitive advantage? it's really a devops software company," Puppet, Tech. Rep., April 2018. [Online]. Available: https://puppet.com/resources/case-study/ambit-energy

[14] J. Schwarz, "Code Smell Detection in Infrastructure as Code," https://www.swc.rwth-aachen.de/thesis/code-smell-detection-infrastructure-code/, 2017, [Online; accessed 02-July-2019].

[15] E. van der Bent, J. Hage, J. Visser, and G. Gousios, "How good is your puppet? an empirically defined and validated quality model for puppet," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2018, pp. 164–174.

[16] A. Rahman and L. Williams, "Characterizing defective configuration scripts used for continuous deployment," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, April 2018, pp. 34–45.

[17] A. Rahman, A. Partho, D. Meder, and L. Williams, "Which factors influence practitioners' usage of build automation tools?" in *Proceedings of the 3rd International Workshop on Rapid Continuous Software Engineering*, ser. RCoSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 20–26. [Online]. Available: https://doi.org/10.1109/RCoSE.2017..8

[18] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, "A systematic mapping study of infrastructure as code research," *Information and Software Technology*, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584918302507

[19] A. Rahman and L. Williams, "Source code properties of defective infrastructure as code scripts," *Information and Software Technology*, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584919300965

[20] O. Hanappi, W. Hummer, and S. Dustdar, "Asserting reliable convergence for configuration management scripts," *SIGPLAN Not.*, vol. 51, no. 10, pp. 328–343, Oct. 2016. [Online]. Available: http://doi.acm.org/10.1145/3022671.2984000

[21] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[22] J. Saldana, *The coding manual for qualitative researchers*. Sage, 2015.

[23] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960. [Online]. Available: http://dx.doi.org/10.1177/001316446002000104

[24] National Institute of Standards and Technology, "Security and privacy controls for federal information systems and organizations," https://www.nist.gov/publications/security-and-privacy-controls-federal-information-systems-and-organizations-including-0, 2014, [Online; accessed 04-July-2019].

[25] T. Ylonen and C. Lonvick, "The secure shell (ssh) protocol architecture," 2006.

[26] M.-A. Storey, J. Ryall, R. I. Bull, D. Myers, and J. Singer, "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*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 251–260. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368123

[27] P. Mutaf, "Defending against a denial-of-service attack on tcp." in *Recent Advances in Intrusion Detection*, 1999.

[28] E. Rescorla, "Http over tls," 2000.

[29] L. of Cryptography and S. S. (CrySyS), "skywiper (a.k.a. flame a.k.a. flamer): A complex malware for targeted attacks," Laboratory of Cryptography and System Security, Budapest, Hungary, Tech. Rep., May 2012. [Online]. Available: http://www.crysys.hu/skywiper/skywiper.pdf

[30] B. den Boer and A. Bosselaers, "Collisions for the compression function of md5," in *Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology*, ser. EUROCRYPT '93. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1994, pp. 293–304. [Online]. Available: http://dl.acm.org/citation.cfm?id=188307.188356

[31] X. Wang and H. Yu, "How to break md5 and other hash functions," in *Proceedings of the 24th Annual International Conference on Theory and Applications of Cryptographic Techniques*, ser. EUROCRYPT'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 19–35. [Online]. Available: http://dx.doi.org/10.1007/11426639_2

[32] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/*icomment: Bugs or bad comments?*/," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 145–158. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294276

[33] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni, "Identifying the characteristics of vulnerable code changes: An empirical study," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 257–268. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635880

[34] S. Bugiel, S. Nurnberger, T. Poppelmann, A.-R. Sadeghi, and T. Schneider, "Amazon IA: When elasticity snaps back," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 389–400. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046753

[35] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, ser. MSR '05. New York, NY, USA: ACM, 2005, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083147

[36] B. Chen and Z. M. Jiang, "Characterizing and detecting anti-patterns in the logging code," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, May 2017, pp. 71–81.

[37] R. C. Bose, "On the construction of balanced incomplete block designs," *Annals of Eugenics*, vol. 9, no. 4, pp. 353–399, 1939.

[38] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, pp. 159–174, 1977. [Online]. Available: http://www.jstor.org/stable/2529310

[39] A. Rahman, M. Rahman, C. Parnin, and L. Williams, "Dataset for Security Smells for Ansible and Chef Scripts Used in DevOps," 7 2019. [Online]. Available: https://figshare.com/s/9f6f1c5bfa6cca9b9214

[40] R. Krishna, A. Agrawal, A. Rahman, A. Sobran, and T. Menzies, "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*, ser. ICSE-SEIP '18. New York, NY, USA: ACM, 2018, pp. 306–315. [Online]. Available: http://doi.acm.org/10.1145/3183519.3183548

[41] A. Agrawal, A. Rahman, R. Krishna, A. Sobran, and T. Menzies, "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*, ser. ICSE-SEIP '18. New York, NY, USA: ACM, 2018, pp. 245–253. [Online]. Available: http://doi.acm.org/10.1145/3183519.3183549

[42] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, pp. 1–35, 2017. [Online]. Available: http://dx.doi.org/10.1007/s10664-017-9512-6

[43] A. Rahman, A. Agrawal, R. Krishna, and A. Sobran, "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*, ser. SWAN 2018. New York, NY, USA: ACM, 2018, pp. 8–14. [Online]. Available: http://doi.acm.org/10.1145/3278142.3278149

[44] J. C. Kelly, J. S. Sherif, and J. Hops, "An analysis of defect densities found during software inspections," *Journal of Systems and Software*, vol. 17, no. 2, pp. 111 – 117, 1992. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0164121292900893

[45] O. H. Alhazmi and Y. K. Malaiya, "Quantitative vulnerability assessment of systems software," in *Annual Reliability and Maintainability Symposium, 2005. Proceedings.*, Jan 2005, pp. 615–620.

[46] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2–13, Jan 2007.

[47] M. Meli, M. R. McNiece, and B. Reaves, "How bad can it git? characterizing secret leakage in public github repositories," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/how-bad-can-it-git-characterizing-secret-leakage-in-public-github-repositories/

[48] T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some code smells have a significant but small effect on faults," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, pp. 33:1–33:39, Sep. 2014. [Online]. Available: http://doi.acm.org/10.1145/2629648

[49] M. V. Mantyla and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," *Empirical Softw. Engg.*, vol. 11, no. 3, pp. 395–431, Sep. 2006. [Online]. Available: http://dx.doi.org/10.1007/s10664-006-9002-8

[50] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 101–110. [Online]. Available: http://dx.doi.org/10.1109/ICSME.2014.32

[51] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the usability of cryptographic apis," in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 154–171.

[52] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, Sept 1975.

[53] E. Barker, "Guideline for using cryptographic standards in the federal government: Cryptographic mechanisms," National Institute of Standards and Technology, Gaithersburg, Maryland, Tech. Rep., August 2016. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-175b.pdf

[54] Y. Acar, C. Stransky, D. Wermke, C. Weir, M. L. Mazurek, and S. Fahl, "Developers need support, too: A survey of security advice for software developers," in *2017 IEEE Cybersecurity Development (SecDev)*, Sept 2017, pp. 22–26.

Akond Rahman Akond Rahman is an assistant professor at Tennessee Tech University. His research interests include DevOps, Software Security, and Software Analytics. He graduated with an M.Sc. in Computer Science and Engineering from University of Connecticut and a B.Sc. in Computer Science and Engineering from Bangladesh University of Engineering and Technology. He won the Microsoft Open Source Challenge Award in 2016, the ACM SIGSOFT Doctoral Symposium Award at International Conference On Software Engineering (ICSE) in 2018, and the ACM SIGSOFT Distinguished Paper Award at ICSE in 2019. During his PhD tenure he has collaborated with industry practitioners from ABB, IBM, Mozilla, RedHat, and others. To know more about his work visit https://akondrahman.github.io/

Rayhanur Rahman Md Rayhanur Rahman is a doctoral student in Department of Computer Science, North Carolina State University, USA. He has done bachelors and masters in software engineering from University of Dhaka. His broad research interest includes analysis of software source code, contextual analysis of security issues in software systems and distributed computing.

Chris Parnin Chris Parnin is an assistant professor at North Carolina State University. His research spans the study of software engineering from empirical, human-computer interaction, and cognitive neuroscience perspectives, publishing over 60 papers. He has worked in Human Interactions in Programming groups at Microsoft Research, performed field studies with ABB Research, and has over a decade of professional programming experience in the defense industry. His research has been recognized by the SIGSOFT Distinguished Paper Award at ICSE 2009, Best Paper Nominee at CHI 2010, Best Paper Award at ICPC 2012, IBM HVC Most Influential Paper Award 2013, CRA CCC Blue Sky Idea Award 2016. He research has been featured in hundreds of international news articles, Game Developer's Magazine, Hacker Monthly, and frequently discussed on Hacker News, Reddit, and Slashdot.

Laurie Williams Laurie Williams is a Distinguished Professor in the Computer Science Department of the College of Engineering at North Carolina State University (NCSU). Laurie is a co-director of the NCSU Science of Security Lablet sponsored by the National Security Agency. Lauries research focuses on software security; agile software development practices and processes; software reliability, and software testing and analysis.