# Detecting and Characterizing Propagation of Security Weaknesses in Puppet-based Infrastructure Management

Akond Rahman, *Member, IEEE* and Chris Parnin *Member, IEEE*

**Abstract**—Despite being beneficial for managing computing infrastructure automatically, Puppet manifests are susceptible to security weaknesses, e.g., hard-coded secrets and use of weak cryptography algorithms. Adequate mitigation of security weaknesses in Puppet manifests is thus necessary to secure computing infrastructure that are managed with Puppet manifests. A characterization of how security weaknesses propagate and affect Puppet-based infrastructure management, can inform practitioners on the relevance of the detected security weaknesses, as well as help them take necessary actions for mitigation. We conduct an empirical study with 17,629 Puppet manifests with **Taint** Tracker for **Pup**pet Manifests (**TaintPup**). We observe 2.4 times more precision, and 1.8 times more F-measure for TaintPup, compared to that of a state-of-the-art security static analysis tool. From our empirical study, we observe security weaknesses to propagate into 4,457 resources, i.e, Puppet-specific code elements used to manage infrastructure. A single instance of a security weakness can propagate into as many as 35 distinct resources. We observe security weaknesses to propagate into 7 categories of resources, which include resources used to manage continuous integration servers and network controllers. According to our survey with 24 practitioners, propagation of security weaknesses into data storage-related resources is rated to have the most severe impact for Puppet-based infrastructure management.

**Index Terms**—configuration as code, devops, devsecops, empirical study, infrastructure as code, puppet, static analysis.

## 1 Introduction

Infrastrcture as code (IaC) is the practice of automatically managing computing infrastructure, such as continuous integration servers, production web servers, load balancers, or data storage, typically provisioned on public cloud services [34]. Use of IaC languages, such as Puppet has yielded benefits for information technology (IT) organizations. For example, Ambit Energy, an energy distribution company, increased their deployment frequency by a factor of 1,200 using Puppet [58]. KPN, a Dutch telecommunications company, uses Puppet manifests to manage its 10,000 servers [59]. Use of Puppet helped KPN in regulatory compliance and faster resolution of customer service requests [59].

Despite reported benefits, Puppet manifests can contain security weaknesses [64], [67], which can leave computing infrastructure susceptible to large-scale security attacks. In recent years, security weaknesses, such as hard-coded passwords and use of weak cryptography algorithms, have been a contributing factor in multiple high-profile security incidents. For example, hard-coded passwords were leveraged to gain unauthorized access to Uber's servers, which resulted in data exposure for 57 million customers and 600,000 Uber drivers [48], [71]. In another incident, use of weak encryption algorithms for Amazon S3 data storage allowed malicious attacks to access over a billion health records [22], [23].

The above-mentioned examples showcase the need for proactively detecting security weaknesses in software artifacts used to provision and manage computing infrastructure. Puppet manifests are no exception. SLIC, a state-of-the-art security static analysis tool for Puppet, based on pattern matching [16], [64], [67] can be used to detect security weaknesses in Puppet manifests. Unfortunately, SLIC can be prone to reporting false positives [12], which deter practitioners from adopting or taking actions [20], [37], [68]. For example, Bhuiyan and Rahman [12] found that SLIC to generate 1,560 false positives for 2,764 Puppet manifests. Generation of such false positives would render security weakness detection impractical for practitioners, leaving security weaknesses unmitigated in Puppet manifests. Thus, enhanced static analysis tools are required for detecting security weaknesses in Puppet manifests.

Examples from the open source software (OSS) domain provide clues on how Puppet-related security static analysis can be enhanced. Let us consider two Puppet manifests from an OSS repository [43] that use SHA1, a weak encryption algorithm. According to the Common Weakness Enumeration (CWE), use of weak encryption algorithms, such as SHA1 is "*dangerous because a determined attacker may be able to break the algorithm and compromise whatever data has been protected*" [50]. In one manifest (Figure 1a), the weakly encrypted password propagates into a Puppet resource [41] used for storing user authentication, potentially allowing malicious users to access the server. In the second manifest (Figure 1b), although a SHA1 hash is created, it never propagates into a resource to manage any relevant computing infrastructure. From these examples, we can see that a security static analysis tool for Puppet must *first* detect potential security weaknesses, and then *second*, determine propagation into the underlying resources used to manage relevant computing infrastructure. This second step is crucial, as, not only does it help eliminate false



*Manuscript received 2022.*

```
define apache::htpasswd_user({
  ...
  $real_password = htpasswd_sha1($password)
  ...
  file_line{"htpasswd_for_${real_site}":
       ensure => $ensure,
       path => $real_path,
       line => "${username}:${real_password}",
  }
}
```

Use of SHA1 to encrypt a password used in line # 8 for resource `file_line`

a

```
class couchdb::base {
  ...
  $pw = [ $::couchdb::admin_pw ]
  $sha1 = str_sha1($pw)
  ...
  exec { 'couchdb_restart':
    command => $restart_command,
    path => ['/bin', '/usr/bin',],
    subscribe => File['/etc/couchdb/local.d/admin.ini',
                      '/etc/couchdb/local.ini'],
    refreshonly => true
  }
}
```

Use of SHA1 to encrypt a password that is never used

b

Fig. 1: OSS Puppet manifests that use SHA1, a weak encryption algorithm. In one manifest (Figure 1a), the hash propagates into a resource to setup a password file. In the second manifest (Figure 1b), although a SHA1 hash is created, it never propagates into any resources used to manage computing infrastructure.

positives (e.g., the weakness in Figure 1b), but information about propagation and associated resources can also help in determining relevance of detected weaknesses [75].

Puppet uses a state-based approach for infrastructure management [41], which necessitates development of novel static analysis tools for detection of security weakness propagation. Puppet infers the desired infrastructure state from the Puppet manifest with code constructs, such as resources [41]. Puppet will identify the differences between the existing and desired infrastructure states, and only apply changes if there are differences between desired and infrastructure states. Along with applying a state-based infrastructure management, Puppet allows multiple categories of information flows with code constructs, such as nodes, modules, and resources [41]. Not all of these code constructs are used to manage infrastructure. Tracking all information flows will not only be computation intensive, but also lead to generating false positives. Therefore, for detecting security weakness propagation a static analysis tool must separate and track information flows that are only used for managing infrastructure.

In this paper, we construct **Taint** Tracker for **Pup**pet Manifests (***TaintPup***), which applies Puppet-specific information flow analysis that helps us to detect and understand how security weaknesses propagate into infrastructure managed with Puppet resources. TaintPup leverages resources, i.e., code elements that are pivotal to account for state-based infrastructure management, along with the corresponding information flows. With TaintPup, we conduct an empirical study with 17,629 Puppet manifests mined from 336 OSS repositories. We quantify how propagation detection improves identification of security weaknesses. Next, we investigate the categories of resources into which security weaknesses propagate. Finally, we survey 24 practitioners, and observe their perceptions for the identified resource categories. Dataset and source code used in our paper is available online [60].

We are the first to perform an investigation of how infrastructure is impacted by security weaknesses. We observe by accounting for Puppet's state-based approach we can improve detection accuracy of security weaknesses and identify the impacted infrastructure. The state-based infrastructure approach has not been accounted in prior work to perform security static analysis of Puppet manifests.

Specifically, we answer the following research questions:

- **RQ**$_1$: *How does propagation detection improve security weakness identification in Puppet manifests?*
- **RQ**$_2$: *How frequently do security weaknesses propagate into resources?*
- **RQ**$_3$: *What are the resource categories into which security weaknesses propagate?*
- **RQ**$_4$: *What are the practitioner perceptions of the identified resources into which security weaknesses propagate?*

**Contributions**: We list our contributions as follows:

- A static analysis tool called TAINTPUP that identifies the resources into which security weaknesses propagate; and
- An empirical evaluation of the resources into which security weaknesses propagate.



## 2 TAINT TRACKER FOR PUPPET: TAINTPUP

In this section, first, we provide background information of Puppet manifests. Next, describe the construction of TaintPup. Third, we provide the methodology and the answer to RQ$_1$.

### 2.1 Background

In the case of Puppet, configuration for the infrastructure of interest is specified using configuration files called 'manifests'. Puppet manifests have a '.pp' extension. Each manifest can include a class, which acts as a placeholder for code constructs, such as resources and variables. Classes in Puppet manifests are different from classes used in object-oriented programming languages. One manifest can include multiple classes, and multiple other code elements, such as multiple variables and multiple resources. A resource is a code element that is used make changes to the desired infrastructure. A resource includes multiple attributes.

We use Figure 2 to further illustrate an example Puppet manifest. This is an example of a Puppet manifest that includes a class called 'example'. The manifest includes two resources used to create two files namely 'a.txt' and 'b.txt'. Both files are located in the '/tmp/' directory. Both resources are of type 'file', and include four attributes namely, 'ensure', 'owner', 'group', 'mode', and 'content'. The manifest also includes a variable called 'strData', which is used by the attribute 'content' in both resources.

```
class example{

$strData = "This is an example text file."

file { '/tmp/a.txt':
  ensure => present,
  owner => 'root',
  group => 'root',
  mode => '0755',
  content => $strData,
   }

file { '/tmp/b.txt':
  ensure => present,
  owner => 'root',
  group => 'root',
  mode => '0755',
  content => $strData,
   }
}
```

Fig. 2: An example Puppet manifest.

Puppet uses a state reconciliation approach to manage computing infrastructure. State reconciliation is defined as the approach of managing computing infrastructure by comparing the inferred state and the desired state with inventory discovery, inventory communication, state comparison, and provisioning. Upon execution of this manifest, Puppet will first query the infrastructure to check for availability of the infrastructure, and then the availability of the files 'a.txt' and 'b.txt'. Puppet will only make the changes specified using the two resources if the two files with specified configurations are not present in the infrastructure. Such process of Puppet is referred to as the 'state-based approach for infrastructure management', as Puppet first will check if the desired state of infrastructure is already in place. If not, Puppet will make necessary changes as specified by resources.

## 2.2 Construction of TaintPup

We use this section to describe TaintPup's construction. We construct TaintPup by accounting the following properties unique to Puppet:

- ***State-based Infrastructure Management***: Puppet uses a state-based approach where manifests are developed in a manner so that it reaches a desired state [41]. During execution *first* Puppet will infer what is the desired infrastructure state from the Puppet manifest. *Second*, Puppet will identify the differences between the existing and desired infrastructure states, and only apply changes if there are differences between desired and infrastructure states. Puppet uses resources and attributes to query the desired infrastructure state so that it can determine what changes need to be made to reach the desired infrastructure state.
  Puppet is a rich language with a variety of code elements. It is pivotal to gain an understanding of which code elements actually are used to manage computing infrastructure. If we do not account for the code elements that are used for Puppet-based infrastructure management, then an automated technique may have explored all possible options on how a security weakness can be used amongst code elements. This approach is not only computationally expensive, but also is susceptible to generate false positives.
  A tool that aims to detect security weakness propagation must account for Puppet's state-based approach for infrastructure management. To address this issue, we construct data dependence graphs, where we track if a security weakness propagates, and affects infrastructure management as described in Section 2.2.3.
- ***Infrastructure-oriented Information Flow***: Puppet allows for multiple categories of information flows with code constructs, such as nodes, classes, modules, and resources [41]. However, not all of these code constructs are used to manage infrastructure. Tracking all information flows will not only be resource intensive, but also lead to generating false positives. To address this challenge we perform three activities:
  - Syntax Analysis: As described in Section 2.2.1, TaintPup performs syntax analysis by applying code element extraction, expression classification, and membership preservation of attributes.
  - Security Weakness Identification: As described in Section 2.2.2, TaintPup applies rule matching to limit the scope of the information flows that need to tracked.
  - Taint Tracking via Data Dependence Graph: As described in Section 2.2.3, TaintPup track the information flows for code elements that constitute a security weakness with data dependence graphs. In this manner, TaintPup only reports a security weakness if that security weakness is being used by a resource.



### 2.2.1 Syntax Analysis

As of June 2022, SLIC [64] is the state-of-art security static analysis tool for Puppet [16]. While Rahman et al. [64] reported SLIC to have an average precision of 0.99, Bhuiyan and Rahman [12] reported SLIC to generate 1,560 false positives for 2,764 Puppet manifests. Reasons for false positives generated by SLIC can be attributed to two types of parsing-related limitations [64]: *(i) code element parsing:* while parsing code elements, SLIC generates false positives by identifying functions as hard-coded secrets. For example, SLIC identifies `$admin_password = pick($access_hash['password'])` as a hard-coded secret, even though (`pick()`) is a function, and not a hard-coded secret; and *(ii) value parsing:* SLIC has limitations in parsing values assigned to Puppet variables. For example, SLIC fails to identify `db_admin_password=undef` as a false positive because it parses `undef` as a string. `undef` in Puppet is not a string, and is actually equivalent to that of `NIL` in Ruby [41]. TaintPup applies the following syntax analysis to account for the above-mentioned limitations of SLIC.

**Code Element Extraction**: *First*, TaintPup uses 'puppet parser dump (PPD)' [41] to identify non-comment code elements in a Puppet manifest. The benefit of using PPD is that with PPD output, TaintPup does not have to apply heuristics for code element extraction, contrary to SLIC [64]. PPD does not provide any API methods, which necessitates

additional pre-processing. PPD converts a Puppet manifest to a single string of tokens where the types of each token is identified. Using stack-based parsing [7] TaintPup extracts tokens and their types from the PPD output. For example, PPD will parse `$db_user='dbadmin'` as `(=($db_user 'dbadmin'))`, which in turn will be used by TaintPup to determine that `$db_user` is a variable, and the value 'dbadmin' is assigned to the variable. Using this step, TaintPup extracts attributes, resources, and variables. Each identified attribute is provided an unique identifier based on the resource and the manifest it appears in.

**Expression Classification**: In Puppet, an expression is an attribute or a variable to which a value is assigned directly, or indirectly, e.g., via a variable or a function [41]. TaintPup identifies three types of expressions: string, function, and parameter. An attribute or a variable that is directly assigned a string value is called a string expression. An attribute or a variable that is assigned a value using a function call is a function expression. A variable that is used as a class parameter, and is directly assigned a string value is called a parameter expression. For example, `$db_user='dbadmin'` is a string expression, `$admin_password = pick($access_hash['password'])` is a function expression, and `$workers='1'` in `class($workers='1'){...}` is a parameter expression. By identifying these expressions TaintPup mitigates SLIC's limitations related to code element and value parsing.

**Membership Preservation of Attributes**: A single Puppet manifest can contain resources, where each resource can have $>= 1$ attributes [41]. Furthermore, an attribute with the same name can appear for multiple resources [41]. TaintPup uses hash maps to map an attribute to its corresponding resource and manifest. These hash maps are later used to construct data dependence graphs (DDGs) that is discussed in Section 2.2.3. Figure 3 shows how the three attributes presented in Figure 1a ('ensure', 'path', 'line') are mapped to their resource `file_line`.

#### 2.2.2 Security Weakness Identification

A security weakness is an insecure coding pattern, which (i) has a mapping with the Common Weakness Enumeration (CWE) database entry [49], and (ii) is actually being used by a code element. In the context of Puppet manifests security weaknesses are recurring coding patterns, which have a mapping with a CWE entry, and is actually being used by a Puppet-specific code element. According to this definition, the insecure coding pattern in Figure 1a is a true positive, and the insecure coding pattern in Figure 1b is a false positive.

We use the following steps. *First, TaintPup applies rule matching* on extracted function expressions, parameter expressions, string expressions, and resources from Section 2.2.1 to identify six security weakness categories. The rules used by TaintPup are listed in Table 1, where we provide the names, definitions, and rules for each security weakness category. Patterns used by the rules are listed in Table 2. All rules and patterns are provided by Rahman et al. [64]. *Second, TaintPup identifies variables*, which are used in any security weakness that belongs to any of the following categories: admin by default, empty password, hard-coded secret, invalid IP address binding, use of HTTP without TLS, and use of weak cryptography algorithms. TaintPup also keeps track of attributes for which a security weakness appears. The security weaknesses categories are identified from prior work conducted by Rahman et al. [64]. They [64] used a qualitative analysis technique called open coding [69], where they manually inspected 1,726 Puppet manifests, and derived a category of security weaknesses that can be mapped to CWE entries. Rahman et al. [64] also developed a tool that can identify security weaknesses. However, their tool SLIC [64], does not perform adequate syntax analysis and state-based information flow analysis, which in turn generates false positives. We address this limitation by constructing and evaluating TaintPup.

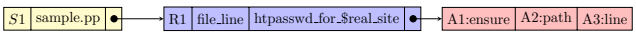


Fig. 3: An example of how TaintPup maps attributes listed in Figure 1 to their resource (`file_line`).

#### 2.2.3 Taint Tracking via Data Dependence Graph

TaintPup applies information flow analysis by constructing DDGs similar to prior research [45]. A DDG is a directed graph, which consists of a set of nodes and a set of edges. DDGs used by TaintPup applies information flow analysis by identifying def-use relationships [7].

Def-use relationships leverage the definition of reachability [7]. A weakness $x$ reaches another attribute or variable $y$, if $y$ uses $x$ and there are no other code elements between $x$ and $y$ that changes the value of $x$. We use Table 3 to demonstrate reachability. In the first row, `$magnum_proto` ="http:" is a string expression with the variable `$magnum_proto`. This string expression is an instance of HTTP without TLS. As shown in the 'Reachability' column, `$magnum_proto` reaches the `url` attribute, because between `$magnum_proto` and `url` there exists no code element that changes the value of `$magnum_proto`. On the other hand, as shown in line#1,#2 of the second row, `$magnum_proto` does not reach `url` as the value of `$magnum_proto` is changed from 'http' to 'ftp:'.

Each DDG has three types of nodes: taint, intermediate, and sink. Taint nodes correspond to variables with security weaknesses that are identified in Section 2.2.2. Sink nodes correspond to attributes used within a resource. Intermediate nodes are non-taint and non-sink nodes that are reachable from one or multiple taint nodes. By construction a DDG will include at least one taint node and at least one sink node. A DDG includes $>= 0$ intermediate nodes.

### 2.3 Methodology for $RQ_1$

We answer $RQ_1$ by computing the detection accuracy of TaintPup with metrics, such as precision [77]. An increase in precision for TaintPup compared to that of SLIC will provide evidence on how detection propagation aids in security weakness identification. We use the following steps in this regard:

#### 2.3.1 Dataset Construction

We use OSS repositories mined from GitHub, GitLab, and three IT organizations who use Puppet manifests to man-

TABLE 1: Rules to Detect Security Weaknesses

| Category | Definition | Rule |
|---|---|---|
| Admin by default | Administrative privileges for users by default [64] | $(isParameter(x)) \wedge (isAdmin(x.name) \wedge isUser(x.name))$ |
| Empty password | Using a string of length zero for a password [64] | $(isAttribute(x) \vee isVariable(x)) \wedge ((length(x.value) == 0 \wedge isPassword(x.name))$ |
| Hard-coded secret | Revealing sensitive information (user names, passwords, private keys) [64] | $(isAttribute(x) \vee isVariable(x)) \wedge (isUser(x.name) \vee isPassword(x.name) \vee isPvtKey(x.name)) \wedge (length(x.value)>0)$ |
| Invalid IP address binding | Assigning '0.0.0.0' as an IP address [64] | $((isVariable(x) \vee isAttribute(x)) \wedge (isInvalidBind(x.value))$ |
| Use of HTTP without TLS | Using HTTP without TLS [64] | $(isAttribute(x) \vee isVariable(x)) \wedge (isHTTP(x.value))$ |
| Use of weak crypto. algo. | Using MD5 and SHA1 [64] | $(isFunction(x) \wedge usesWeakAlgo(x.name))$ |

TABLE 2: Patterns Used by Rules in Table 1

| Function | String Pattern |
|---|---|
| $isAdmin()$ [64] | 'admin' |
| $isHTTP()$ [64] | 'http:' |
| $isInvalidBind()$ [64] | '0.0.0.0' |
| $isPassword()$ [64] | 'pwd', 'pass', 'password' |
| $isPvtKey()$ [64] | '[pvt\|priv]+*[cert\|key\|rsa\|secret\|ssl]+' |
| $isUser()$ [64] | 'user' |
| $usesWeakAlgo()$ [64] | 'md5', 'sha1' |

TABLE 3: An Example to Demonstrate Reachability

| Coding Pattern | Reachability |
|---|---|
| 1. `$magnum_proto = ''http:''`<br>2. `package{'sample'`<br>`ensure => '4.2.1-5.fc25',`<br>`url =>`<br>`$magnum_proto//''localhost:8888''`<br>`}` | `$magnum_proto = ''http:''` reaches `url` |
| 1. `$magnum_proto = ''http:''`<br>2. `$magnum_proto = ''ftp:''`<br>3. `package{'sample'`<br>`ensure => '4.2.1-5.fc25',`<br>`url =>`<br>`$magnum_proto//localhost:8888`<br>`}` | `$magnum_proto = ''http:''` does not reach `url` |

age their computing infrastructure: Mozilla, Openstack, and Wikimedia Commons. The purpose of using datasets from five sources is to showcase the generalizability of our empirical study. Relying only one dataset could have made our empirical study susceptible to external validity, which we mitigate using five datasets. Of these five datasets, repositories mined from GitHub and GitLab are reflective of Puppet manifest development across a wide range of organizations. The datasets related to Mozilla, Openstack, and Wikimedia are repositories respectively, mined from the repository archives hosted by Mozilla, Openstack, and Wikimedia Commons. As these organizations are involved in creating different software services we assume to observe differences in the constructed datasets. We do observe in Table 6 that the frequency of security weaknesses is different across the datasets. Furthermore, the frequency of impacted resources and resource category is different across datasets as described in Sections 4.1 and 4.2.

The software engineering research community has heavily relied on open-source software repositories to conduct empirical analysis. However, while conducting these research studies, researchers have realized that open-source repositories are not always reflective of real-world software development [13], [38], [52]. Researchers [13], [38], [52] have advocated for application of heuristics to filter out repositories prior to using them for research. Researchers [6], [40], [52] have leveraged a set of attributes to filter OSS GitHub repositories reflective of professional software development. These attributes include count of Puppet manifests [62], count of commits per month [52], and count of contributors [6], [40]. Taking motivation from prior work [52], we apply the following filtering criteria:

- *Criterion-1*: The proportion of Puppet manifests is $>=$ 10%. A repository can include multiple types of files including Puppet manifests. In prior work Jiang and Adams [36] reported that within repositories Puppet manifests co-locate with source code files and test code files. Keeping this observation into account, similar to prior work [62], [63], [64], [65], we use the heuristic of a repository to include at least 10% Puppet manifests to determine a repository to include sufficient amount of Puppet manifests for analysis.
- *Criterion-2*: The repository is not a copy of another to avoid duplicates.
- *Criterion-3*: Count of contributors is $>=$ 10. Similar to prior work [64], [65], we use this criterion to filter repositories used for personal purposes, such as coursework.
- *Criterion-4*: Lifetime of the repository is $>=$ 1 month. Using this criterion, we filter repositories with short lifetime. We measure lifetime by calculating the difference between the last commit date and the creation date for the repository.
- *Criterion-5*: The repository has $>=$ 25 commits to filter repositories with limited activity.
- *Criterion-6*: The repository has $>=$ 2 commits per month. Munaiah et al. [52] used this threshold to identify mature OSS GitHub repositories.

We collect the Mozilla, Openstack, and Wikimedia repositories from their corresponding public repository databases (Mozilla [51], Openstack [54], Wikimedia [79]). We use Google BigQuery [32] to download OSS repositories hosted on GitHub that use Puppet. We use the GitLab API [25] to mine OSS repositories hosted on GitLab. Table 4 summarizes how many repositories are filtered using our cri-

teria. We download 336 repositories by cloning the master branches on October 2021. Attributes of collected repositories is available in Table 5. We collect 17,629 Puppet manifests from these 336 OSS repositories. We do not remove any manifests as we wanted to run our empirical analysis on unaltered data obtained from OSS repositories. In that manner, we assume to obtain a "close to reality" view of the state of security weakness propagation in Puppet manifests.

TABLE 4: Repository Filtering

| | **GitHub** | **GitLab** | **Mozilla** | **Openstack** | **Wiki.** |
|---|---|---|---|---|---|
| | 3,405,303 | 1,659 | 1,594 | 2,262 | 2,509 |
| Criterion-1 | 18,187 | 38 | 2 | 96 | 13 |
| Criterion-2 | 17,872 | 38 | 2 | 96 | 13 |
| Criterion-3 | 856 | 30 | 2 | 94 | 13 |
| Criterion-4 | 770 | 30 | 2 | 90 | 11 |
| Criterion-5 | 675 | 30 | 2 | 90 | 11 |
| Criterion-6 | 241 | 25 | 2 | 61 | 7 |
| **Final** | 241 | 25 | 2 | 61 | 7 |

#### 2.3.2 Dataset Labeling

We use three steps to perform labeling:

*Step#1-Rater Training*: A first year PhD student, who is not an author of the paper, volunteered to participate in labeling all 17,629 Puppet manifests. The rater has an experience of one year in software engineering and cybersecurity. Before applying labeling, we conduct a training session where the rater is mentored by the first author. The training session was conducted in two phases: *first in phase-1.1*, both the first author and the rater independently inspect a randomly-selected set of 500 Puppet manifests. The first author and the rater use a (i) guidebook [60] with names, definitions, and examples of security weakness categories, and (ii) the online documentation of Puppet [41] that describes syntax and information flow in Puppet manifests.

We derive the set of 500 manifests using the concept of purposeful sampling by taking motivation from prior research in software engineering [11]. Purposeful sampling is a popular sampling technique to identify a sample from the population based on one or multiple selection criteria [46], [55]. Upon derivation of the sample the researcher or research team need to manually examine the sample to determine if the criteria are satisfied [46], [55]. In our case, for the set of 500 manifests our criterion is to inspect if all six categories of security weaknesses are represented in the set of manifests. We observe that the selected 500 manifests include multiple instances of the six security weakness categories, which gives us the confidence of that the selected set of 500 manifests is sufficient. With a margin of error of 5%, the confidence level for the sample size of 500 was 98%.

We apply a multi-phase open coding process with two phases following prior work on fault categorization [15]. According to researchers, multi-phase coding is pivotal to gain multiple perspectives, ensure rater reliability, and achieve rater consensus for qualitative analysis [31], [76]. As part of the first phase, we conduct synchronized closed coding where we use the set of 500 randomly selected manifests. Using these manifests, the first author ensured that the rater has the necessary background to conduct full-scale closed coding on the entire set of 17,629 Puppet manifests. As the initial set of 500 manifests included instances of all the six categories of security weaknesses, the rater obtained sufficient background on the six categories of security weaknesses.

The rater and the first author individually determines if a security weakness is in fact true positive by first inspecting if a security weakness exists, and then if the weakness is used by $>= 1$ resources. Upon completion of the inspection process the rater discuss their agreements and disagreements. At this stage the Cohen's Kappa is 0.47, indicating 'moderate' agreement according to Landis and Koch [42]. The misunderstanding occurred due to not comprehending what actually a true positive security weakness is. The rater was further briefed on the fact that to determine a security weakness the rater also needs to find out if the security weakness is actually being used by a resource. *Second in phase-1.2*, upon discussion of their disagreements, the rater and the first author conducted another round of inspection. The process and used materials are similar to that of phase-1.1. At this stage, Cohen's Kappa is 1.0 between the rater and the first author, which gives us the confidence that the rater is equipped with necessary background to label all 17,629 manifests.

*Step#2-Labeling*: Similar to the training session, the rater uses the guidebook and the Puppet online documentation [41] to identify security weaknesses in 17,629 Puppet manifests. As part of the dataset labeling activity, the rater first inspected if in the manifest there exists one or multiple security weaknesses. Next, the rater inspected if each of the identified security weakness propagates into at least one resource. A security weakness is listed as a true positive if it is used by a resource. One manifest can include multiple categories of security weaknesses, and thus the rater can map one Puppet manifest to one or multiple of the six categories. The rater takes 745 hours to complete labeling. Altogether, the rater identifies 4,906 security weaknesses. Count of security weaknesses for the five datasets is presented in Table 6.

*Step#3-Rater Verification*: We use a PhD student in the department, who is not an author of the paper, to verify the rater's labeling. We use a randomly-selected set 500 manifests from our set of 17,629 manifests that is not used in Step#1. Similar to Step#1, the PhD student is provided the guidebook and Puppet's online documentation [41]. Upon completion, we record a Cohen's Kappa of 0.91 between the PhD student and the rater.

#### 2.3.3 Evaluate TaintPup's Detection Performance

We evaluate TaintPup's detection performance by applying the following steps: *first*, we run TaintPup and SLIC on the collected 17,629 manifests. The total execution time for running 17,629 Puppet manifests is 48.9 hours, 10 seconds on average for each manifest. The tool is executed on an Apple M2, 16 GB memory. *Second*, we use three metrics: precision, recall, and F-measure, similar to prior work [56]. Precision refers to the fraction of correctly identified instances among the total identified security weaknesses, as determined by a static analysis tool. Recall refers to the fraction of correctly identified instances retrieved by a static analysis tool over the total amount instances. F-measure is the harmonic mean of precision and recall [77].

TABLE 5: Dataset Attributes

| Attribute | GitHub | GitLab | Mozilla | Openstack | Wiki. | Combined |
|---|---|---|---|---|---|---|
| Total Repos. | 241 | 25 | 2 | 61 | 7 | 336 |
| Total Commits | 599,900 | 1,943 | 14,449 | 42,446 | 16,231 | 674,969 |
| Average Duration (Month) | 241 | 34.2 | 90 | 38.5 | 60.0 | 92.7 |
| Total Puppet Manifests | 11,477 | 883 | 1,613 | 2,952 | 704 | 17,629 |
| Total Puppet LOC | 498,241 | 49,430 | 66,367 | 234,640 | 27,889 | 876,567 |
| Total Distinct Resources | 65,599 | 5,055 | 10,583 | 23,754 | 3,561 | 108,552 |

TABLE 6: Count of Security Weaknesses in Our Datasets

| Category | GitHub | GitLab | Mozilla | Openstack | Wiki. |
|---|---|---|---|---|---|
| Admin by default | 5 | 0 | 0 | 12 | 0 |
| Empty password | 40 | 0 | 2 | 3 | 21 |
| Hard-coded secret | 2,604 | 105 | 145 | 751 | 63 |
| Invalid IP address binding | 31 | 1 | 12 | 62 | 0 |
| Use of HTTP without TLS | 543 | 7 | 5 | 465 | 22 |
| Use of weak crypto. algo. | 3 | 2 | 0 | 2 | 0 |
| **Total** | 3,226 | 115 | 164 | 1,295 | 106 |

### 2.4 Answer to RQ$_1$: TaintPup's Detection Accuracy

We answer **RQ$_1$: How does propagation detection improve security weakness identification in Puppet manifests?** in this section. We report the precision and F-measure for SLIC and TaintPup with Tables 7 and 8. With respect to precision, TaintPup outperforms SLIC for all categories across all datasets. According to Table 7, TaintPup's average precision is 3.3, 2.5, 2.4, 3.4, and 1.5 times higher than that of SLIC respectively, for GitHub, GitLab, Mozilla, Openstack, and Wikimedia. Considering 4,906 security weaknesses across all five datasets, the average precision is 2.4 times higher than that of SLIC. Furthermore, across all five datasets the average F-measure is 1.8 times higher than that of SLIC. We observe a recall of 1.0 for both SLIC and TaintPup for all six categories. This shows TaintPup's ability to detect all identified security weaknesses with higher precision than that of SLIC, without reducing recall.

**Evaluation of SLIC and TaintPup on Bhuiyan et al.'s Dataset**: We also compare TaintPup and SLIC's detection accuracy using the dataset labeled by Bhuiyan et al. [12]. The results are provided in Table 9. We observe SLIC and TaintPup to have the same recall. The average precision and F-measure is respectively, 2.2 and 1.6 times higher for TaintPup compared to that of SLIC.

**Evaluation of SLIC and TaintPup on SLIC Dataset**: We also compare TaintPup and SLIC's detection accuracy using the dataset provided by Rahman et al. [64]. In the replication package provided by provided by Rahman et al. [64] we do not find any labeling. As a result, we recruit a rater to perform labeling on randomly-selected set of 100 Puppet manifests used by Rahman et al. [64]. The rater who participated in the dataset labeling process described in Section 2.3.2, performed labeling for the oracle dataset provided by Rahman et al. [64]. The results are provided in Table 10. We observe SLIC and TaintPup to have the same recall. The average precision and F-measure is respectively, 2.7 and 1.9 times higher for TaintPup than that of SLIC.

**Practitioner Feedback on TaintPup**: We also submit pull requests to collect feedback from practitioners in order to get further validation. In total, we submit pull requests for 100 security weaknesses across 19 OSS repositories. A breakdown of the security weakness categories and corresponding count is presented in Table 11. Of the submitted pull requests we obtain feedback for 1 security weaknesses (response rate=1%).

Our response rate is low, which can be attributed to a lack of monetary incentives [62], [74], practitioners' negative biases for static analysis alerts [37], [64], [65], [66] as well as for submitted bug reports related to security alerts [65]. Survey response rate in cybersecurity and software engineering research can respectively, be as low as 4.1% [66] and 6% [74]. We mitigate the limitation of low response rate for bug reports by conducting semi-structured interviews. We describe the methodology and results of conducted semi-structured interviews as follows:

**Methodology for Conducting Semi-structured Interviews:** We use randomly-selected 100 email addresses and sent emails to all 100 email addresses. For our semi-structured interview we used emails from the repositories that we mined and described in Section 2.3.1. The first author of the paper sent the emails. Upon response and approval, we invited the participants over Zoom. In all, we found 5 interviewees who agree to participate. All interviewees participated via Zoom.



As part of this semi-structured review, we first state the purpose of the interview, demonstrate TaintPup, and then we ask questions. We describe each of these steps below:

*Purpose*: The purpose of the interview is to understand if TaintPup is useful for practitioners to detect security weaknesses.

*Demonstration of TaintPup*: We perform the following activities for demonstration following He et al. [28]'s recommendations:

- Proposition: Proposition corresponds to describing the goal of the semi-structured interview. The goal of the interview is to obtain feedback from practitioners about the usefulness of TaintPup.
- Evidence: Evidence corresponds to the artifacts that are used for the interview. As part of this activity we describe the construction and usage of TaintPup. We also describe verbally the security weakness categories with examples.
- Method of demonstration: As part of this activity we showcase how TaintPup can be executed from the command line. We ran TaintPup on a repository, showed the output it generates, described the execution flow, and walked through the generated comma-separated value (CSV) file by discussing the meaning of each column.

Upon demonstration of TaintPup we ask: "*Do you think TaintPup is useful to detect security weaknesses in Puppet manifests?*". We impose no limit on time to answer these questions. We also allowed the participants to talk about any topics that they think is relevant to the answers of the above-mentioned question.

TABLE 7: Answer to $RQ_1$: Precision of SLIC and TaintPup for Six Security Weakness Categories

| | GitHub | | GitLab | | Mozilla | | Openstack | | Wikimedia | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Category** | **SLIC** | **TaintPup** | **SLIC** | **TaintPup** | **SLIC** | **TaintPup** | **SLIC** | **TaintPup** | **SLIC** | **TaintPup** |
| Admin by default | 0.15 | 0.83 | NA | NA | NA | NA | 0.10 | 0.86 | NA | NA |
| Empty password | 0.06 | 0.93 | NA | NA | 0.30 | 1.00 | 0.08 | 0.75 | 0.75 | 0.84 |
| Hard-coded secret | 0.63 | 0.95 | 0.55 | 0.92 | 0.50 | 0.96 | 0.49 | 0.94 | 0.44 | 0.83 |
| Invalid IP address binding | 0.20 | 0.89 | 0.25 | 1.00 | 0.62 | 0.86 | 0.26 | 0.93 | NA | NA |
| Use of HTTP without TLS | 0.53 | 0.97 | 0.47 | 0.88 | 0.11 | 0.83 | 0.59 | 0.96 | 0.52 | 0.88 |
| Use of weak crypto. algo. | 0.03 | 0.60 | 0.27 | 1.00 | NA | NA | 0.07 | 1.00 | NA | NA |
| **Average** | 0.26 | 0.86 | 0.38 | 0.95 | 0.38 | 0.91 | 0.26 | 0.90 | 0.57 | 0.85 |

TABLE 8: Answer to $RQ_1$: F-measure of SLIC and TaintPup for Six Security Weakness Categories

| | GitHub | | GitLab | | Mozilla | | Openstack | | Wikimedia | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Category** | **SLIC** | **TaintPup** | **SLIC** | **TaintPup** | **SLIC** | **TaintPup** | **SLIC** | **TaintPup** | **SLIC** | **TaintPup** |
| Admin by default | 0.27 | 0.91 | NA | NA | NA | NA | 0.18 | 0.92 | NA | NA |
| Empty password | 0.12 | 0.93 | NA | NA | 0.46 | 1.00 | 0.15 | 0.86 | 0.86 | 0.91 |
| Hard-coded secret | 0.77 | 0.97 | 0.81 | 0.96 | 0.67 | 0.98 | 0.66 | 0.97 | 0.69 | 0.79 |
| Invalid IP address binding | 0.33 | 0.94 | 0.40 | 1.00 | 0.76 | 0.92 | 0.42 | 0.96 | NA | NA |
| Use of HTTP without TLS | 0.69 | 0.98 | 0.62 | 0.93 | 0.21 | 0.91 | 0.75 | 0.98 | 0.68 | 0.94 |
| Use of weak crypto. algo. | 0.05 | 0.75 | 0.54 | 1.00 | NA | NA | 0.07 | 1.00 | NA | NA |
| **Average** | 0.37 | 0.91 | 0.59 | 0.97 | 0.52 | 0.95 | 0.37 | 0.95 | 0.74 | 0.88 |

TABLE 9: Precision, Recall, and F-measure of SLIC and TaintPup for the Dataset Provided by Bhuiyan et al. [12]

| | Precision | | Recall | | F-Measure | |
|---|---|---|---|---|---|---|
| **Category** | **SLIC** | **TaintPup** | **SLIC** | **TaintPup** | **SLIC** | **TaintPup** |
| Admin by default | 0.25 | 0.87 | 1.0 | 1.0 | 0.40 | 0.93 |
| Empty password | 0.66 | 0.98 | 1.0 | 1.0 | 0.80 | 0.99 |
| Hard-coded secret | 0.53 | 0.94 | 1.0 | 1.0 | 0.69 | 0.97 |
| Invalid IP address binding | 0.29 | 0.94 | 1.0 | 1.0 | 0.45 | 0.97 |
| Use of HTTP without TLS | 0.73 | 0.91 | 1.0 | 1.0 | 0.84 | 0.95 |
| Use of weak crypto. algo. | 0.01 | 0.72 | 1.0 | 1.0 | 0.02 | 0.84 |
| **Average** | 0.41 | 0.89 | 1.0 | 1.0 | 0.58 | 0.94 |



TABLE 10: Precision, Recall, and F-measure of SLIC and TaintPup for the Dataset Provided by Rahman et al. [64]

| | Precision | | Recall | | F-Measure | |
|---|---|---|---|---|---|---|
| **Category** | **SLIC** | **TaintPup** | **SLIC** | **TaintPup** | **SLIC** | **TaintPup** |
| Admin by default | 0.11 | 0.81 | 1.0 | 1.0 | 0.20 | 0.90 |
| Empty password | 0.36 | 0.91 | 1.0 | 1.0 | 0.53 | 0.95 |
| Hard-coded secret | 0.65 | 0.88 | 1.0 | 1.0 | 0.79 | 0.94 |
| Invalid IP address binding | 0.19 | 0.84 | 1.0 | 1.0 | 0.32 | 0.91 |
| Use of HTTP without TLS | 0.53 | 0.89 | 1.0 | 1.0 | 0.69 | 0.94 |
| Use of weak crypto. algo. | 0.11 | 0.87 | 1.0 | 1.0 | 0.20 | 0.93 |
| **Average** | 0.32 | 0.86 | 1.0 | 1.0 | 0.48 | 0.93 |

TABLE 11: Categories of Security Weaknesses for Submitted Pull Requests

| **Category** | **Count** |
|---|---|
| Admin by default | 2 |
| Empty password | 10 |
| Hard-coded secret | 71 |
| Invalid IP address binding | 1 |
| Use of HTTP without TLS | 12 |
| Use of weak crypto. algo. | 4 |
| **Total** | 100 |

**Results from Semi-structured Interviews:** All five participants agreed that TaintPup is useful to detect security weaknesses. Their experience in Puppet manifest development is listed in Table 12 with the 'Experience' column. The 'Usefulness' column corresponds to practitioners' perceived usefulness of TaintPup. In the interviews, the participants also provided additional feedback. For example, P1 stated "*Security is something I have meetings on everyday and I would say that in general it [TaintPup] would be very valuable. Tying this to an external standard [CWE] makes it very valuable. Very cool!*" P2 added "*Yes, it is useful for developers. Let me know by email if I can further connect to you to more people inside [the organization] so that we can get it integrated in our pipeline*". P4 stated "*I would use it in my pipeline. I liked the tool honestly. It helps, it helps!*".

## 3 Methodology for Empirical Study

Methodology to answer $RQ_2$, $RQ_3$, and $RQ_4$ is described below.

TABLE 12: Results from Semi-structured Interviews

| ID | Experience (Years) | Job Title | Usefulness |
|---|---|---|---|
| P1 | 6 | Senior Developer | YES |
| P2 | 7 | Consultant | YES |
| P3 | 7 | SRE | YES |
| P4 | 5 | DevOps Engineer | YES |
| P5 | 1 | Developer | YES |

### 3.1 Methodology to Answer $RQ_2$

We use $RQ_2$ to characterize how frequently security weaknesses propagate into resources. Such characterization can help us understand how many resources are being impacted by one or multiple security weaknesses. We answer $RQ_2$ by *first* reporting the total count of resources in each dataset into which security weaknesses propagate. *Second*, with Equation 1 we compute 'Impacted Resource (%)', i.e., the proportion of resources in each dataset for which $>= 1$ security weakness propagates. We use 'Impacted Resource', as resource is the fundamental unit to specify configurations in order to manage a computing infrastructure [41]. If we can demonstrate empirical evidence that the identified security weaknesses are actually used by resources, then practitioners will understand how security weaknesses are used, and the resources they are used in. *Third*, we report the minimum, median, and maximum number of resources in a manifest into which a security weakness propagate. To answer $RQ_2$, we use TaintPup as it allows us to identify security weaknesses used by attributes within resources with the help of DDGs.

$$\text{Impacted Resource(\%)} = \frac{\text{\# of resources in which } >= 1 \text{ security weakness propagates}}{\text{total \# of unique resources in the dataset}} * 100\% \quad (1)$$

### 3.2 Methodology to Answer $RQ_3$

We conduct a categorization of affected resources to gain further understanding of the resources affected by security weaknesses. We apply the following steps: *first*, we identify affected attributes and resources from the output of TaintPup. *Second*, we identify the titles and types from the affected resources. *Third*, we apply open coding [69] with titles and types of affected resources. Open coding is a qualitative analysis technique that is used to identify categories from structured or unstructured text [69]. *Fourth*, we compute the proportion of resources that belong to a certain category.

Our open coding process can be described as follows: first we derive initial codes by inspecting the names and types of each resource into which at least one security weakness propagates. From those initial codes we derive initial categories based on similarities between the initial codes. Finally, we merge initial categories into one category if there are similarities between initial categories.

We use Figure 4 to demonstrate our open coding process even further. First, we extract initial codes from code snippets. Then from these initial codes we obtain initial categories, which are later merged into a category. For example, from the initial categories `MySQL database management with Puppet resources` and `PostgreSQL database management with Puppet resources`, we derive a category called `Database management with Puppet resources`.

*Rater Verification*: The first author who has 6 years of experience in Puppet development performs open coding. The derived categories are susceptible to rater bias, which we mitigate by using a PhD student in the department, who is not an author of the paper. The additional rater was assigned 100 randomly-selected resources for mapping them to the categories identified by the first author. We record a Cohen's Kappa [18] of 0.87 between the PhD student and the first author, indicating 'perfect' agreement [42]. We derive the set of 100 resources using the concept of purposeful sampling, a popular sampling technique to identify a sample from the population based on one or multiple selection criteria [46], [55]. In the case of 100 resources our selection criterion is the presence of six categories of resources. From our manual inspection we observe the six resource categories to appear in the 100 resources. This gives us the confidence that the selected set of 100 resources is sufficient. With a margin of error of 5%, the confidence level for the sample size of 100 resources was 95%.

### 3.3 Methodology to Answer $RQ_4$

We answer $RQ_4$ by conducting an online survey with practitioners who have developed Puppet manifests. We contact these practitioners via e-mails, which we mine from the 336 OSS repositories reported in Section 2.3.1. We randomly select 250 e-mail addresses, which we use to send the surveys. We offer a drawing of two 50 USD Amazon gift cards as an incentive for participation following Smith et al. [73]'s recommendations. We conduct the survey from December 2021 to March 2022 following the Internal Review Board (IRB) protocol #2356.

In the survey, we first ask developers about their experience in developing Puppet manifests. Next, we describe each of the identified resource categories with definitions and examples. We then ask questions related to perceived frequency and severity: *first*, we ask "*How frequently do you think the identified resource categories are affected by security weaknesses?*" Survey participants used a five-item Likert scale to answer this question: 'Not at all frequent', 'Rarely', 'Somewhat frequently', 'Frequently', and 'Highly frequent'. *Second*, we ask "*What is the severity of the identified resource categories into which security weaknesses appear?*" To answer this question, survey participants used the following five-item Likert scale: 'Not at all severe', 'Low severity', 'Moderately severe', 'Severe', and 'Highly severe'. We use a five-item Likert scale for both questions following Kitchenham and Pfleeger's guidelines [39]. Furthermore, following Kitchenham and Pfleeger [39]'s advice we apply the following actions before deploying the survey: (i) provide an estimate of completion time, (ii) provide explanations related to the purpose of the study, (iii) provide survey completion instructions, and (iv) provide explanations confirming preservation of confidentiality. The survey questionnaire is included in our verifiability package [60].

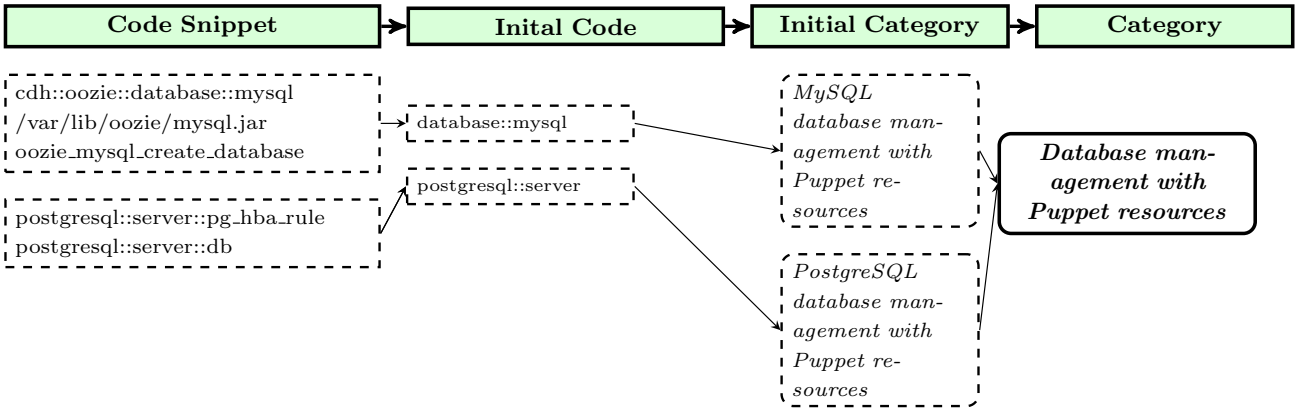


Fig. 4: An example to demonstrate our open coding process used to answer $RQ_3$.

## 4 Empirical Findings

We provide answers to $RQ_2$, $RQ_3$, and $RQ_4$ as follows:

### 4.1 Answer to $RQ_2$: Propagation Frequency

In this section, we provide answer to **$RQ_2$: How frequently do security weaknesses propagate into resources?** Altogether, we identify 4,906 security weaknesses to propagate into 4,457 distinct resources. The count of resources into which weaknesses propagate is: 2,945 for GitHub, 104 for GitLab, 167 for Mozilla, 1,128 for Openstack, and 113 for Wikimedia. Considering all datasets, security weaknesses propagate into 4.1% of 108,552 resources. The 'Impacted Resource (%)' column shows the proportion of resources into which $>= 1$ security weaknesses propagate. For example, for GitHub, we observe 3,872 security weaknesses to propagate into 4.49% of 65,599 resources. The proportion of affected resources is highest for Openstack. Details are available in Table 13.

TABLE 13: Answer to $RQ_2$: Frequency of Resources Into Which Security Weaknesses Propagate

| | Impacted Resource (%) | | | | |
|---|---|---|---|---|---|
| **Category** | **GitH.** | **GitL.** | **Mozilla** | **Open.** | **Wiki.** |
| Admin by default | 0.01 | NA | 0.00 | 0.05 | NA |
| Empty password | 0.09 | 0.02 | 0.02 | 0.02 | 0.05 |
| Hard-coded secret | 3.55 | 1.82 | 1.30 | 3.25 | 2.22 |
| Invalid IP address binding | 0.05 | 0.02 | 0.17 | 0.19 | 0.00 |
| Use of HTTP without TLS | 0.79 | 0.16 | 0.09 | 1.24 | 0.89 |
| Use of weak crypto. algo. | 0.01 | 0.02 | NA | 0.004 | NA |
| **Combined** | 4.49 | 2.06 | 1.58 | 4.75 | 3.17 |

In Table 14, we report the minimum, median, and maximum number of resources in a manifest into which $>= 1$ security weakness propagates. We observe a security weakness can propagate into as many as 35 distinct resources.

### 4.2 Answer to $RQ_3$: Resource Categories

We provide answers to **$RQ_3$: What are the resource categories into which security weaknesses propagate?** by first describing the resource categories in Section 4.2.1. Next, we report frequency of resource categories in Section 4.2.2.

TABLE 14: Answer to $RQ_2$: Resource Frequency (Minimum, Median, Maximum)

| **Category** | **GitHub** | **GitLab** | **Mozilla** | **Openstack** | **Wiki.** |
|---|---|---|---|---|---|
| Admin by default | 1, 1, 2 | NA | 0, 0, 0 | 1, 1, 1 | NA |
| Empty password | 1, 2, 5 | 1, 1, 1 | 1, 1, 1 | 1, 1, 1 | 1, 1, 1 |
| Hard-coded secret | 1, 1, 35 | 1, 1, 6 | 1, 1, 6 | 1, 1, 12 | 1, 1, 4 |
| Invalid IP address binding | 1, 1, 2 | 1, 1, 1 | 1, 3, 5 | 1, 1, 3 | 0, 0, 0 |
| Use of HTTP without TLS | 1, 1, 7 | 1, 3, 4 | 1, 2.5, 4 | 1, 1, 6 | 1, 1, 6 |
| Use of weak crypto. algo. | 1, 1, 1 | 1, 1, 1 | NA | 1, 1, 1 | NA |
| **Total** | 1, 1, 35 | 1, 1, 6 | 1, 1, 6 | 1, 1, 12 | 1, 1, 6 |



#### 4.2.1 Description of Resource Categories

We identify 7 categories of resources into which security weaknesses propagate. We describe each category with examples as follows.

I-***Communication Platforms***: Resources used to manage communication platforms, such as Discourse [1] and Slack [2].

*Example*: Listing 1 shows how a hard-coded user name is used to manage Slack contacts in line# 9. The hard-coded username is `$slack_username = 'Icinga'`, which is later used by the resource `icinga::slack_contact`.

II-***Containerization***: Resources used to manage containers.

*Example*: Listing 2 shows an example of a resource that is used to perform authentication for the `Magnum` container service [3]. We observe an instance of insecure HTTP for `$magnum_protocol`, which is later used by two attributes in the `magnum` resource.

In Listing 2 there is one security weakness, `$magnum_protocol='http'`, which propagates into the resource `magnum '::magnum::keystone::authtoken':`. TaintPup reports `$magnum_protocol='http'` as a true positive security weakness. `$magnum_url` does not propagate into the resource and therefore is not a true positive security weakness.

1. https://www.discourse.org/
2. https://slack.com/
3. https://wiki.openstack.org/wiki/Magnum

```
1 $notify_slack = false,
2 $notify_graphite = true,
3 $slack_channel = undef,
4 $slack_username = 'Icinga',
5 ...
6 icinga::slack_contact { 'slack_search_team':
7     slack_webhook_url       =>
      ↪  $slack_webhook_url,
8     slack_channel           =>
      ↪  '#govuk-searchandnav',
9     slack_username          =>
      ↪  $slack_username,
10    icinga_status_cgi_url   =>
      ↪  $slack_icinga_status_cgi_url,
11    icinga_extinfo_cgi_url =>
      ↪  $slack_icinga_extinfo_cgi_url,
12 }
```

Listing 1: A hard-coded secret propagating into a resource used to manage Slack.

```
1 $magnum_protocol = 'http'
2 ...
3 $magnum_url = "${magnum_protocol}://${magnum_host}:$magnum_port/v1"
4 magnum { '::magnum::keystone::authtoken':
5     auth_uri => "${magnum_protocol}://${magnum_host}:5000/v3",
6     auth_url => "${magnum_protocol}://${magnum_host}:35357",
7     ...
8 }
```

Listing 2: An instance of insecure HTTP propagating into a resource used to manage Magnum-based containers.

III-***Continuous Integration***: Resources used to manage infrastructure needed to implement the practice of continuous integration (CI), with tools, such as Jenkins [35]. CI tools integrate code changes by automatically compiling, building, and executing test cases upon submission of code changes [21].

*Example*: In Listing 3 an instance of empty password propagates into a resource to setup configurations for Jenkins. As shown in line #9, the empty password instance `$jenkins_management_password = ''` is used to construct `$security_opt_params` using `join`, a Puppet function used to concatenate strings [41]. Later with the `exec` resource, `$security_opt_params` is used to manage configurations for a Jenkins-based CI infrastructure.

IV-***Data Storage***: Resources used to manage data storage systems, such as MySQL servers, PostgreSQL servers, and Memcached.

*Example*: Listing 4 shows an instance of empty password used by a resource to manage a MySQL database. The `mysql::db` resource uses `$database_password=''` with the `password` attribute.

V-***File***: Resources used to manage files by performing file-related operations, such as reading, writing, or deleting a file. We observe security weaknesses to propagate in Puppet-defined resources, such as `file`, and custom resources.

*Example*: Listing 5 shows how SHA1 is used to encrypt a password with the `htpasswd_sha1` function. The encrypted password is assigned to `$nagiosadmin_pw`, which is later used to manage a file with the `File['nagios_htpasswd']` resource, as shown in line#5.

VI-***Load Balancers***: Resources used to manage load balancers, such as HAPRoxy [27]. Load balancing is used to distribute network or application traffic across multiple servers [14].

*Example*: Listing 6 shows an example of a security weakness propagating into a resource used to manage HAProxy. `$vip` is an instance of an invalid IP address, which is used by `$api_vip_orig` and `$discovery_vip_orig` respectively, in lines #8 and 14. Both `$api_vip_orig` and `$discovery_vip_orig` will be assigned '0.0.0.0' with `$vip` through the execution of the else block as both `$api_server_vip` and `$discovery_server_vip` is assigned `undef`, which is `false` when used as Boolean. `$api_vip_orig` and `$discovery_vip_orig` are respectively, used in lines #17 and #21 to manage HAProxy services. Listing 6 is an example that illustrates TaintPup's ability to detect the propagation of one security weakness into multiple resources.

```
1 class jenkins::master (
2   ...
3   $jenkins_management_password      = '',
4   ...
5   $security_opt_params = join([
6       'set_security_password',
7       "'${jenkins_management_login}'",
8       "'${jenkins_management_email}'",
9       "'${jenkins_management_password}'",
10      "'${jenkins_management_name}'",
11      "'${jenkins_ssh_public_key_contents}'",
12      "'${jenkins_s2m_acl}'",
13  ], ' ')
14  ...
15  exec { 'jenkins_auth_config':
16    require   => [
17      File["${jenkins_libdir}/jenkins_cli.groovy"],
18      Package['groovy'],
19      Service['jenkins'],
20    ],
21    command   => join([
22      '/usr/bin/java',
23      "-jar ${jenkins_cli_file} -s",
        ↪  "${jenkins_proto}://${jenkins_address}:"
24      "${jenkins_port}",
25      "groovy
        ↪  ${jenkins_libdir}/jenkins_cli.groovy",
26      $security_opt_params,
27    ], ' '),
28    tries     => $jenkins_cli_tries,
29    ...
30  }
```

Listing 3: An empty password propagating into a resource used to manage Jenkins.

```
1 class gerrit::mysql(
2   ...
3   $database_password = '',
4 ) {
5   mysql::db { $database_name:
6     ...
7     password => $database_password,
8     host     => 'localhost',
9     grant    => ['all'],
10    ...
11  }
12  ...
13 }
```

Listing 4: An instance of empty password propagating into a resource used to manage a MySQL database.

```
1 $nagiosadmin_pw    =
  ↪  htpasswd_sha1($nagios_hiera['nagiosadmin_pw'])
2 $nagios_hosts      = $nagios_hiera['hosts']
3 File['nagios_htpasswd'] {
4   source  => undef,
5   content => "nagiosadmin:${nagiosadmin_pw}",
6   mode    => '0640',
7 }
```

Listing 5: An instance of SHA1 usage propagating into a resource used to manage a file.

```
1 $vip                      = '0.0.0.0',
2 $api_server_vip           = undef,
3 $discovery_server_vip     = undef,
4 ...
5 if $api_server_vip {
6    $api_vip_orig = $api_server_vip
7 } else {
8   $api_vip_orig = $vip
9 }
10
11 if $discovery_server_vip {
12   $discovery_vip_orig = $discovery_server_vip
13 } else {
14   $discovery_vip_orig = $vip
15 }
16 rjil::haproxy_service { 'api':
17     vip               => $api_vip_orig,
18     ...
19 }
20 rjil::haproxy_service { 'discovery':
21     vip               => $discovery_vip_orig,
22     ...
23 }
```

Listing 6: An instance of invalid IP address propagating into a resource used to manage HAProxy, a load balancer.

VII-*Networking*: Resources used to manage network-related functionalities, such as setting up firewalls and network controllers, as well as managing virtual local area networks (VLANs) and virtual private networks (VPNs).

*Example*: Listing 7 shows an example of a hard-coded password propagating into a resource used for management of network infrastructure. The resource is used to manage the Open Network Operating System (ONOS) controller [53]. The hard-coded password `$password = 'karaf'` is used by `$dashboard_desc`, which is later used to construct `$json_hash`. In line #10, `$json_hash` is used by `$json_messsage`. Later, as shown in line#12, the `exec` resource uses $json_message to execute a command in line#12 in order to create an ONOS dashboard link.

### 4.2.2 Frequency of Affected Resource Categories

Findings from Table 15 show that security weaknesses are in fact used for managing infrastructure, such as CI, container, and data storage infrastructure. For example, for GitHub 69% of identified hard-coded secrets are used to manage CI-based infrastructure. A complete breakdown is available in Table 15, where we provide a mapping between each security weakness and resource categories into which security weaknesses propagate. 'CI', 'Comm', 'Container', 'Data', 'File', 'Load', and 'Network' respectively refers to continuous integration, communication platforms, containerization,

```
1 notice(' ONOS MODULAR: onos-dashboard.pp')
2 ...
3 $password = 'karaf'
4 ...
5 $dashboard_desc = "Onos dashboard interface.
  ↪  Default credentials are ${user}/${password}"
6
7 $json_hash = { title       => $dashboard_name,
8                description => $dashboard_desc,
9                url         => $dashboard_link, }
10 $json_message = $json_hash
11 exec { 'create_dashboard_link':
12   command => "/usr/bin/curl -H 'Content-Type:
    ↪  application/json' -X POST -d
    ↪  '${json_message}'
    ↪  https://${master_ip}:8000/api/clusters/..."
13 }
```

Listing 7: A hard-coded password propagating into a resource used to manage ONOS.

data storage, file, load balancer, and network. A resource category name is followed by the proportion of security weaknesses that propagate into resources for that category. 'NA' indicates a security weakness to not propagate into a resource for a certain category. The percentage of affected resource categories is listed in Figure 5. For example, the total count of affected resources by security weaknesses is 2,945 for the Github dataset, of which 0.1% are used to manage communication platforms. Our findings provide a cautionary tale on the state of Puppet manifest security, as we observe identified security weaknesses to propagate into resources for infrastructure management, which in turn leaves computing infrastructure susceptible to security attacks.

One possible explanation of the quantified propagation is related to how Puppet manifests are used for infrastructure provisioning. Puppet is used to implement the practice of IaC. Accordingly, we observe Puppet manifests to provision a wide range of computing infrastructure, such as data storage, network infrastructure, and communication platforms.

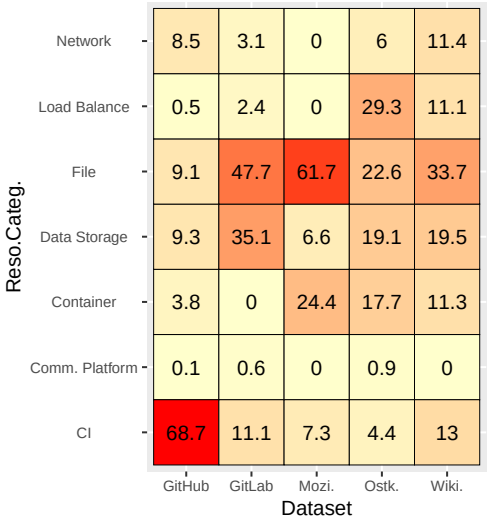


Fig. 5: Answer to $RQ_3$: Percentage of Affected Resources.

TABLE 15: Answer to $RQ_3$: Mapping of Resource Categories and Security Weaknesses

| Category | GitHub | GitLab | Mozilla | Openstack | Wikimedia |
|---|---|---|---|---|---|
| Admin by default | Data:100% | NA | NA | Container:11.1%, Data:27.8%, File:5.5%, Load:27.8%, Network:27.8% | NA |
| Empty password | CI:62.0%, Data:32.4%, File:5.6% | File:100% | File:100% | Data:40.0%, Load:60.0% | Data:100% |
| Hard-coded secret | CI:69%, Container:3.6%, Comm:0.1%, Data:8.8%, File:9.3%, Load:0.3%, Network:8.9% | File:45%, Data:55% | CI:10.0%, Container:21.9%, Data:0.3%, File:67.8% | CI:3.5%, Container:18.7%, Comm:1.1%, Data:38.4%, File:31.6%, Load:2.6%, Network:4.1% | CI:11.1%, Container:1.6%, Data:78.6%, File:5.5%, Load:0.8%, Network:2.4% |
| Invalid IP address binding | CI:2.4%, Container:7.3%, Data:43.9%, File:2.4%, Load:21.9%, Network:22.1% | Data:100% | Network:100% | Container:18.6%, Data:40.6%, File:6.8%, Load:28.9%, Network:5.1% | NA |
| Use of HTTP without TLS | CI:49.6%, Container:22.2%, Data:16.5%, File:3.9%, Load:5.7%, Network:2.1% | Container:100% | Container:100% | CI:7.1%, Container:15.6%, Comm:0.7%, Data:31.1%, File:4.8%, Load:31.0%, Network:9.7% | CI:2.1%, Data:85.4%, File:12.5% |
| Use of weak crypto. algo. | File:50%, Data:50% | File:100% | NA | Data:100% | NA |

### 4.3 Answer to $RQ_4$: Practitioner Perception

In this section, we provide answer to **$RQ_4$: What are the practitioner perceptions of the identified resources into which security weaknesses propagate?**. From our survey, we obtain 24 responses in total. The survey respondents were distributed across 14 repositories. A complete breakdown of the respondents' experience in Puppet manifest development is provided in Table 16.



TABLE 16: Survey Respondents' Experience in Puppet Manifest Development

| Experience | Respondent count |
|---|---|
| $< 1$ year | 11 |
| $1 - 2$ years | 1 |
| $3 - 4$ years | 3 |
| $>= 5$ years | 9 |

In Figures 6 and 7 we respectively, report practitioner perceptions for frequency and severity of the identified resource categories. The x and y-axis respectively presents the percentage of survey participants and resource categories. For example, from Figure 6 we observe 25% of the total survey respondents to identify containerization as a resource category for which security weaknesses frequently or highly frequently propagate.

From Figure 6, we observe survey respondents to perceive CI management to be most frequently affected by security weaknesses. Such perception is congruent with the GitHub-related findings presented in Figure 5, where we observe resources related to CI to be the most frequent category. Furthermore, based on Figure 5 we observe the proportion of resources related to management of communication platforms to be $< 1.0\%$. From Figure 7, we observe propagation of security weaknesses for data storage management to be perceived as most severe.

Figures 6 and 7 show nuanced perspectives from practitioners. For example, while 88% of respondents perceive

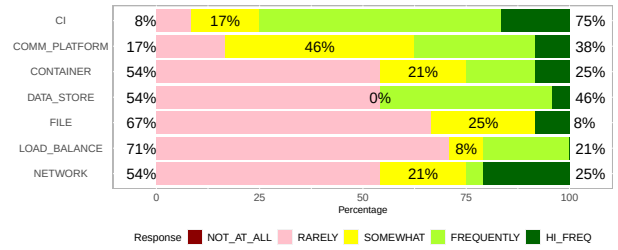


Fig. 6: Answer to $RQ_4$: Practitioner perception of frequency for identified resource categories.

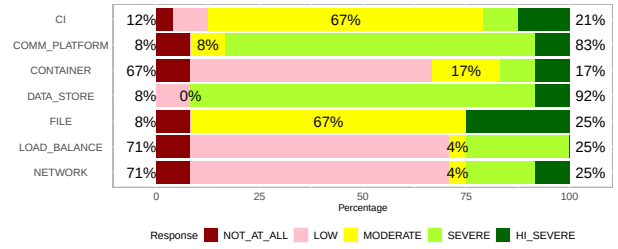


Fig. 7: Answer to $RQ_4$: Practitioner perception of severity for identified resource categories.

CI-related resources to be frequently impacted by security weaknesses, 12% of total respondents find security weaknesses in CI-based management to have severe or highly severe impact. Also, 92% of the respondents identify security weakness propagation for data storage-related resources to be severe or highly severe, but only 46% of the respondents perceive such propagation to be frequent.

## 5 Discussion

We discuss the implications of our paper in this section.

```
1  $username = 'root'  <-- Use of a hard-coded user name in line # 18 for
2                          resource users::root::account
3  $group = $::operatingsystem ? {
4      Darwin => wheel,
5      default => root
6  }
7
8  $home = $::operatingsystem ? {
9      Darwin => '/var/root',
10     # $rlocalpath is set by the winrootlp.rb in the
           ↪ shared module
11     Windows => $::rlocalpath,
12     default => '/root'
13 }
14
15 ...
16     'users::root::account':
17         stage => users,
18         username => $username,
19         group => $group,
20         home => $home;
21 }
```

Fig. 8: An example to demonstrate the usability of TaintPup. TaintPup automatically detects that `$username = 'root'` is a hard-coded user name, and used by the resource called `users::root::account`.

### 5.1 Implications Related to Practitioner Actionability

Our results in Table 7 show TaintPup to have higher precision than that of SLIC, which is a state-of-the-art security static analysis tool for Puppet. Low precision static analysis tool contribute to a lack of actionability, which in turn results in abandonment of static analysis tool usage [10], [30], [37]. Unlike SLIC, TaintPup generates fewer false positives. On average, TaintPup's precision is 2.4 times higher than that of SLIC. We attribute TaintPup's precision improvement to the advanced syntax analysis, and application of information flow analysis. As better precision is correlated with increased actionability for static analysis tools [29], [68], TaintPup's precision improvement can help practitioners take actions to mitigate security weaknesses.

Another utility of TaintPup is its capability to report the flow of a detected security weakness, and whether or not it is being used by a resource. Figure 8 shows the utility of TaintPup. TaintPup automatically detects that `$username = 'root'` is a hard-coded user name, and used by the resource called `users::root::account`. In this manner, TaintPup detects a security weakness, which is a true positive. Unlike SLIC, TaintPup reports the name and location of a manifest, the resources affected by a security weakness, and the attribute used by the resource into which the security weakness propagate. Such capability gives practitioners the ability to assess if a detected security weakness is relevant or not. We recommend use of information flow analysis to detect security weaknesses in Puppet manifests because it (i) identifies resources that are affected by security weaknesses, and (ii) reduces false positives.

### 5.2 Survey-related Implications

Implications of $RQ_4$ are:

**Severity-related Perceptions:** Findings reported in Section 4.3 show practitioners to not identify security weakness propagation for all resource categories to be severe. According to Figure 7, practitioners perceive propagation of security weaknesses to be least severe for CI and container infrastructure management. However, these perceptions could leave unmitigated security weaknesses during management of CI and containers, which in turn could be used by malicious users to perform cryptomining attacks [44]. Existence of security weaknesses in CI infrastructure resulted in the Codecov incident, which impacted 29,000 customers, and breached hundreds of customer networks [1], [2], [4]. According to Table 15, security weaknesses, such as hard-coded secrets propagate into container-related resources, which in turn can cause container escape, where a container user is able to nullify container isolation and access unauthorized resources [47]. Container escapes motivated malicious users conduct security attacks on container-based infrastructure, as many as 17,358 attacks in 18 months [3], [5].

**Frequency-related Perceptions:** Our frequency-related findings in Section 4.3 show a disconnect between what practitioners perceive, and empirical results. While practitioners perceive file-related resources to be least frequently affected by security weaknesses according to Figure 6, these resources are most frequently affected for GitLab, Mozilla, and Wikimedia as shown in Figure 5. These findings suggest a lack of practitioner awareness on how frequently security weaknesses affect Puppet-based infrastructure management, which can be mitigated through the use of TaintPup.

***Implications Related to Prioritizing Inspection Efforts:*** Our empirical study has implications for prioritizing inspection efforts as well. While conducting security focused code reviews, practitioners can focus on the resources for which TaintPup reports a security weakness. In this manner, instead of inspecting all resources, with the help of TaintPup practitioners can inspect a smaller set of resources.



### 5.3 Future Work

We discuss opportunities for future work:

#### 5.3.1 *Improvement Opportunities for TaintPup*

TaintPup can be extended so that practitioners themselves can specify the sinks to track security weaknesses in Puppet manifests. Currently, TaintPup uses attributes in resources as sinks, which could be limiting because a security weakness can be used by a code snippet deemed important by practitioners, but is not an attribute.

TaintPup can be extended to detect more categories of security weaknesses. If a new security weakness category is derived, then the weakness instances can be abstracted into rules, and these rules can be integrated with TaintPup. The DDG construction and querying process will remain unchanged.

Let us consider the example of default ports in this regard. As a hypothetical example, if default port is identified as a security weakness category, then the corresponding rule would be $(isAttribute(x) \vee isVariable(x)) \wedge isPort(x) \wedge hasDefaultValue(x.value)$. This rule can be integrated into TaintPup by adding one function, which will do pattern matching to implement $hasDefaultValue(x.value)$ and $isPort(x)$. There will be no change in the DDG construction and querying process.

#### 5.3.2 Security-focused Information Flow Analysis for Other IaC Languages

Security-focused information flow analysis for other languages is an opportunity for future work. Such analysis will require an understanding of what code elements are used to manage infrastructure in other languages. For example, with respect to syntax and semantics Puppet is different from Ansible [65], which requires information flow analysis tools tailored for Ansible manifests. One approach could be use of single static assignments (SSAs) [7] for detection of security weakness propagation. However, deriving SSAs for IaC languages, such as Ansible and Puppet is challenges as the underlying compilers do not provide SSAs directly.

#### 5.3.3 Severity of Security Weaknesses

Our results presented in Figure 7 lay the groundwork for future research related to the severity of security weaknesses in Puppet manifests. While we acknowledge that practitioners perceptions related to severity is important, triangulation of these perceptions is also required to strengthen the empirical foundations of secure development of Puppet manifests. In particular, empirical studies should consider existing concepts, such as the Common Vulnerability Scoring System (CVSS) [70] to triangulate severity of security weaknesses and their potential impact for Puppet-managed computing infrastructure.

#### 5.3.4 Towards Actionable Repair of Detected Security Weaknesses

Our semi-structured interview provides the groundwork to conduct further research on how to generate repairs for security weaknesses so that practitioners are more motivated to take actions on the detected weaknesses. For example, researchers through semi-structured interviews can ask what possible strategies that they would have taken for the detected weaknesses. The semi-structured interview can also trigger discussions, such as use of secret management tools, and empirically validating relevant best practices [61].

## 6 Threats to Validity

**Conclusion Validity**: TaintPup builds DDGs leveraging def-use chains [7], which may not capture all types of information flow. For example, if a hard-coded password is provided as a command line input or as a catalog [41], then TaintPup will not report a security weakness. An example of a command that a practitioner would write on the terminal to provide a hard-coded password is "`puppet resource user <USERNAME> ensure=present managehome=true password='<PASSOWRD-STRING>'`". The rater who identified security weaknesses did not find any such instance where a hard-coded password is provided as input to a Puppet manifest from the command line. In order to find such instance, the rater would have to get access to the commands that are being typed on the command line or terminal by a practitioner. Unfortunately, these commands do not appear in a manifest, and therefore the rater was not able to identify such security weaknesses. We still acknowledge that the rater's bias could impact the dataset derivation process, which we mitigate using rater verification.

We acknowledge that other code elements exist that is used to manage computing infrastructure. Our results could have been further improved by incorporating such code elements.

**Construct Validity**: In Section 2.3.2, when determining security weaknesses the rater may have implicit biases that could have affected the labeling for the five datasets. Furthermore, the rater might be biased by the security weaknesses included in the sample of 500 manifests. We mitigate this limitation by allocating a rater who is not the author of the paper, and also by performing rater verification. Furthermore, TaintPup leverages PPD to perform parsing, which can yield false positives and false negatives.

**External Validity**: Our empirical study is susceptible to external validity as our analysis is limited to datasets collected from OSS repositories. TaintPup can generate false positives and false negatives for datasets not used in the paper, which in turn can influence results presented in Sections 4.1 and 4.2.

## 7 Related Work

Our paper is related to prior research on Puppet-related code elements that are indicative of quality concerns. Sharma et al. [72], Bent et al. [78], and Rahman and Williams [8] in separate studies identified Puppet-related code elements that are indicative of defects in Puppet manifest. Analysis of specific defects, such as security defects has garnered interest amongst researchers too. By mining OSS repositories Rahman et al. [62] found absence of Puppet code elements to cause security defects. Existence of security defects, such as security weaknesses were further confirmed by Rahman et al. [64], where they identified seven categories of security weaknesses. They further replicated the study in another paper [65], where they observed security weaknesses in Puppet manifests to also appear for Ansible manifests. Rahman et al. [64]'s paper was also replicated by Hortlund [33], who reported the security weakness density to be less than that of reported by Rahman et al., due to false positives generated by SLIC. Bhuiyan and Rahman [12] reported similar observations: they manually inspected 2,764 Puppet manifests, and documented SLIC to generated 1,560 false positives. The closest work in spirit is the research conducted by Rahman et al. [64]. They [64] used a qualitative analysis technique called open coding [69], where they manually inspected 1,726 Puppet manifests, and derived a category of security weaknesses that appear in Puppet manifests. Rahman et al. [64] also developed a tool that can identify security weaknesses. However, their tool SLIC [64], does not perform adequate syntax analysis and track how a security weakness propagates into resources. We address this limitation by constructing and evaluating TaintPup.

Our paper is also related to prior research that has applied taint tracking for quality analysis of Android, Java, and Python applications. Xia et al. [80] used taint tracking to build AppAudit. Using AppAudit, they [80] found most data leaks to be caused by third-party advertising modules. Gibler et al. [24] performed taint tracking to identify 57,299

privacy leaks in 7,414 Android apps. Arzt et al. [9] applied alias-based taint tracking to construct FlowDroid so that leaks are detected in 500 Android apps. Mahmud et al. [45] used taint tracking to identify Android apps that violate Payment Card Industry (PCI) compliance standards. Gordon et al. [26] applied taint analysis to detect inter component communication (ICC) leaks in 24 Android apps. Java-specific taint tracking tools have also been proposed. Chin and Wagner [17] conducted character level taint tracking to detect vulnerabilities in Java web-based applications. Conti and Russo [19] constructed a Python-based taint tracking tool to identify vulnerabilities in Python applications. Peng et al. [57] used taint tracking to verify integrity of Python applications. However, none of these tools are applicable for Puppet manifests as they do not account for Puppet's state-based infrastructure management approach as well as code elements unique to Puppet.

From the above-mentioned discussion we observe propagation of security weaknesses in Puppet manifests to be an under-explored research area, as none of the above-mentioned papers investigate how security weaknesses impact Puppet-based infrastructure management. We address this research gap by constructing TaintPup, and then we use TaintPup to conduct an empirical study.

## 8 Conclusion

While IaC scripts, such as Puppet manifests have yielded benefits for managing computing infrastructure at scale, these manifests include security weaknesses, such as hard-coded passwords and use of weak cryptography algorithms. To detect and characterize security weaknesses propagation for Puppet-based infrastructure management, we have constructed TaintPup, using which we conduct an empirical study with 17,629 Puppet manifests. We observe TaintPup to have 2.4 times more precision compared to that of SLIC, a state-of-the-art security static analysis tool for Puppet. Our empirical study shows security weaknesses to propagate into 4,457 resources, where a single weakness can propagate into as many as 35 distinct resources. Furthermore, we observe security weaknesses to propagate into a variety of resources, e.g., resources used to manage CI and container-based infrastructure. Our survey-related findings indicate a disconnect between developer perception and empirical characterization of security weakness propagation. Such disconnect further highlights the importance of using TaintPup in Puppet manifest development as it can automatically identify resources that are affected by security weaknesses.



## Acknowledgments

We thank the PASER group at Auburn University for their valuable feedback. This research was partially funded by the U.S. National Science Foundation (NSF) Award # 2247141, Award # 2310179, and the U.S. National Security Agency (NSA) Award # H98230-21-1-0175.

## References

[1] Ax Sharma , "Securing CI/CD pipelines: 6 best practices," https://www.csoonline.com/article/3624577/securing-cicd-pipelines-6-best-practices.html, 2021, [Online; accessed 12-Feb-2022].

[2] Joseph Menn and Raphael Satter , "Codecov hackers breached hundreds of restricted customer sites - sources," https://www.reuters.com/technology/codecov-hackers-breached-hundreds-restricted-customer-sites-sources-2021-04-19/, 2021, [Online; accessed 13-Feb-2022].

[3] Kevin Townsend , "Attacks Against Container Infrastructures Increasing, Including Supply Chain Attacks," https://www.securityweek.com/attacks-against-container-infrastructures-increasing-including-supply-chain-attacks, 2021, [Online; accessed 15-Feb-2022].

[4] Raphael Satter , "US investigators probing breach at code testing company Codecov," https://www.reuters.com/technology/us-investigators-probing-breach-san-francisco-code-testing-company-firm-2021-04-16/, 2021, [Online; accessed 14-Feb-2022].

[5] Team Nautilus , "Attacks in the Wild on the Container Supply Chain and Infrastructure," https://info.aquasec.com/hubfs/Threat%20reports/, 2021, [Online; accessed 15-Feb-2022].

[6] 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

[7] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, principles, techniques," *Addison wesley*, vol. 7, no. 8, p. 9, 1986.

[8] R. Akond and W. Laurie, "Source code properties of defective infrastructure as code scripts," *Information and Software Technology*, 2019.

[9] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 259–269. [Online]. Available: https://doi.org/10.1145/2594291.2594299

[10] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 1–8. [Online]. Available: https://doi.org/10.1145/1251535.1251536

[11] T. Barik, D. Ford, E. Murphy-Hill, and C. Parnin, "How should compilers explain problems to developers?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 633–643.

[12] F. A. Bhuiyan and A. Rahman, "Characterizing co-located insecure coding patterns in infrastructure as code scripts," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering Workshops*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 27–32. [Online]. Available: https://doi.org/10.1145/3417113.3422154

[13] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 2009, pp. 1–10.

[14] V. Cardellini, M. Colajanni, and P. S. Yu, "Dynamic load balancing on web-server systems," *IEEE Internet computing*, vol. 3, no. 3, pp. 28–39, 1999.

[15] Z. Chen, H. Yao, Y. Lou, Y. Cao, Y. Liu, H. Wang, and X. Liu, "An empirical study on deployment faults of deep learning based mobile applications," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 674–685.

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

[17] E. Chin and D. Wagner, "Efficient character-level taint tracking for java," in *Proceedings of the 2009 ACM Workshop on Secure Web Services*, ser. SWS '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 3–12. [Online]. Available: https://doi.org/10.1145/1655121.1655125

[18] 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

[19] J. J. Conti and A. Russo, "A taint mode for python via a library," in *Information Security Technology for Applications*, T. Aura, K. Järvinen, and K. Nyberg, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 210–222.

[20] C. Dimastrogiovanni and N. Laranjeiro, "Towards understanding the value of false positives in static code analysis," in *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*, 2016, pp. 119–122.

[21] P. Duvall, S. M. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007.

[22] C. Fripp, "Data breach alert: Info on millions of seniors leaked online," https://www.komando.com/security-privacy/data-breach-impacts-seniors/803085/, 2021, [Online; accessed 11-Jan-2022].

[23] ——, "Over a billion pharmacy records exposed – What it means for your privacy," https://www.komando.com/security-privacy/billion-pharmacy-records-exposed/793746, 2021, [Online; accessed 11-Jan-2022].

[24] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale," in *International Conference on Trust and Trustworthy Computing*. Springer, 2012, pp. 291–307.

[25] Gitlab, "Gitlab REST API Docs," https://docs.gitlab.com/ee/api/README.html#current-status, 2019, [Online; accessed 16-Dec-2020].

[26] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe." in *NDSS*, vol. 15, no. 201, 2015, p. 110.

[27] HAPROXY, "The Reliable, High Performance TCP/HTTP Load Balancer," http://www.haproxy.org/, 2022, [Online; accessed 22-Jan-2022].

[28] W. He, J. Ding, X. Shen, X. Han, and L. Tang, "A survey on software reliability demonstration," *IOP Conference Series: Materials Science and Engineering*, vol. 1043, no. 3, p. 032008, jan 2021. [Online]. Available: https://doi.org/10.1088/1757-899x/1043/3/032008

[29] S. Heckman and L. Williams, "A model building process for identifying actionable static analysis alerts," in *2009 International Conference on Software Testing Verification and Validation*. IEEE, 2009, pp. 161–170.

[30] ——, "A comparative evaluation of static analysis actionable alert identification techniques," in *Proceedings of the 9th International Conference on Predictive Models in Software Engineering*, ser. PROMISE '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2499393.2499399

[31] G. Hickey and C. Kipping, "A multi-stage approach to the coding of data from open-ended questions." *Nurse researcher*, vol. 4, no. 1, pp. 81–91, 1996.

[32] F. Hoffa, "GitHub on BigQuery: Analyze all the open source code," https://cloud.google.com/blog/products/gcp/github-on-bigquery-analyze-all-the-open-source-code, 2016, [Online; accessed 16-Dec-2020].

[33] A. Hortlund, "Security smells in open-source infrastructure as code scripts: A replication study," 2021.

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

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

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

[37] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 672–681. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486877

[38] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "An in-depth study of the promises and perils of mining github," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2035–2071, 2016.

[39] B. A. Kitchenham and S. L. Pfleeger, *Personal Opinion Surveys*. London: Springer London, 2008, pp. 63–92. [Online]. Available: https://doi.org/10.1007/978-1-84800-044-5_3

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

[42] 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

[43] leapcode, "leapcode/leap_platform," https://github.com/leapcode/leap_platform, 2018, [Online; accessed 26-Dec-2021].

[44] Z. Li, W. Liu, H. Chen, X. Wang, X. Liao, L. Xing, M. Zha, H. Jin, and D. Zou, "Robbery on devops: Understanding and mitigating illicit cryptomining on continuous integration service platforms," in *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2022, pp. 363–378. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46214.2022.00022

[45] S. Y. Mahmud, A. Acharya, B. Andow, W. Enck, and B. Reaves, "Cardpliance: PCI DSS compliance of android applications," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1517–1533.

[46] M. N. Marshall, "Sampling for qualitative research," *Family practice*, vol. 13, no. 6, pp. 522–526, 1996.

[47] A. Martin and M. Hausenblas, *Hacking Kubernetes: Threat-Driven Analysis and Defense*. O'Reilly Media, 2021.

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

[49] MITRE, "CWE-Common Weakness Enumeration," https://cwe.mitre.org/index.html, 2021, [Online; accessed 01-July-2021].

[50] ——, "CWE-327: Use of a Broken or Risky Cryptographic Algorithm," https://cwe.mitre.org/data/definitions/327.html, 2022, [Online; accessed 02-Jan-2022].

[51] Mozilla, "Mozilla Mercurial Repositories Index," https://hg.mozilla.org/build, 2021, [Online; accessed 17-Jan-2021].

[52] 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

[53] ONOS, "ONOS," https://wiki.onosproject.org/display/ONOS/, 2020, [Online; accessed 23-Jan-2022].

[54] Openstack, "OpenStack git repository browser," http://git.openstack.org/cgit/, 2020, [Online; accessed 12-December-2020].

[55] L. A. Palinkas, S. M. Horwitz, C. A. Green, J. P. Wisdom, N. Duan, and K. Hoagwood, "Purposeful sampling for qualitative data collection and analysis in mixed method implementation research," *Administration and policy in mental health and mental health services research*, vol. 42, no. 5, pp. 533–544, 2015.

[56] F. Pauck, E. Bodden, and H. Wehrheim, "Do android taint analysis tools keep their promises?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 331–341. [Online]. Available: https://doi.org/10.1145/3236024.3236029

[57] S. Peng, P. Liu, and J. Han, "A python security analysis framework in integrity verification and vulnerability detection," *Wuhan University Journal of Natural Sciences*, vol. 24, no. 2, pp. 141–148, 2019.

[58] 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

[59] Puppet, "About KPN," https://puppet.com/resources/customer-story/kpn, 2021, [Online; accessed 22-May-2021].

[60] A. Rahman, "Verifiability package for paper," https://figshare.com/s/30a15335e471dfbb2075, 2021, [Online; accessed 20-Feb-2022].

[61] A. Rahman, F. L. Barsha, and P. Morrison, "Shhh!: 12 practices for secret management in infrastructure as code," in *2021 IEEE Secure Development Conference (SecDev)*, 2021, pp. 56–62.

[62] A. Rahman, E. Farhana, C. Parnin, and L. Williams, "Gang of eight: A defect taxonomy for infrastructure as code scripts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 752–764. [Online]. Available: https://doi.org/10.1145/3377811.3380409

[63] A. Rahman, E. Farhana, and L. Williams, "The 'as code'activities: development anti-patterns for infrastructure as code," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3430–3467, 2020.

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

[65] A. Rahman, M. R. Rahman, C. Parnin, and L. Williams, "Security smells in ansible and chef scripts: A replication study," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 1, Jan. 2021. [Online]. Available: https://doi.org/10.1145/3408897

[66] A. Rahman, S. I. Shamim, D. B. Bose, and R. Pandita, "Security misconfigurations in open source kubernetes manifests: An empirical study," *ACM Trans. Softw. Eng. Methodol.*, dec 2022, just Accepted. [Online]. Available: https://akondrahman.github.io/files/papers/tosem-k8s.pdf

[67] A. Rahman and L. Williams, "Different kind of smells: Security smells in infrastructure as code scripts," *IEEE Security Privacy*, vol. 19, no. 3, pp. 33–41, 2021.

[68] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Commun. ACM*, vol. 61, no. 4, p. 58–66, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3188720

[69] J. Saldaña, *The coding manual for qualitative researchers*. Sage, 2015.

[70] K. Scarfone and P. Mell, "An analysis of cvss version 2 vulnerability scoring," in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2009, pp. 516–525.

[71] J. Schwarz, "Hardcoded and Embedded Credentials are an IT Security Hazard – Here's What You Need to Know," https://www.beyondtrust.com/blog/entry/hardcoded-and-embedded-credentials-are-an-it-security-hazard-heres-what-you-need-to-know, 2019, [Online; accessed 02-July-2021].

[72] 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

[73] E. Smith, R. Loftin, E. Murphy-Hill, C. Bird, and T. Zimmermann, "Improving developer participation rates in surveys," in *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, May 2013, pp. 89–92.

[74] ——, "Improving developer participation rates in surveys," in *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, 2013, pp. 89–92.

[75] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford, "Questions developers ask while diagnosing potential security vulnerabilities with static analysis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 248–259. [Online]. Available: https://doi.org/10.1145/2786805.2786812

[76] A. Sweeney, K. E. Greenwood, S. Williams, T. Wykes, and D. S. Rose, "Hearing the voices of service user researchers in collaborative qualitative data analysis: the case for multiple coding," *Health Expectations*, vol. 16, no. 4, pp. e89–e99, 2013.

[77] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining, (First Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005.

[78] 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.

[79] Wikimedia, "Wikimedia Code Review," https://gerrit.wikimedia.org/r/admin/repos, 2021, [Online; accessed 16-Jan-2021].

[80] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu, "Effective real-time android application auditing," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 899–914.




**Akond Rahman** Akond Rahman is an assistant professor at Auburn University. His research interests include DevOps and Secure Software Development. He graduated with a PhD from North Carolina State University, 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 ACM SIGSOFT Doctoral Symposium Award at ICSE in 2018, the ACM SIGSOFT Distinguished Paper Award at ICSE in 2019, the CSC Distinguished Dissertation Award, and the COE Distinguished Dissertation Award from NC State in 2020. He actively collaborates with industry practitioners from GitHub, WindRiver, and others. To know more about his work visit https://akondrahman.github.io/

**Chris Parnin** Chris Parnin is a principal researcher at Microsoft. 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.