UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Static Aplication Security Testing Tools

SCOPE, IMPROVEMENTS AND FUTURE APPLICATIONS

Sofia Oliveira Reis

Supervisor : Rui Maranhão Abreu
Co-Supervisor : João Ferreira

Thesis specifically prepared to obtain the PhD Degree in
Computer Science and Engineering

Draft

11 de outubro de 2023

Abstract

The Objective of this Work ... (English)

Keywords

Keywords (English)

Resumo

O objectivo deste trabalho ... (Português)

Palavras Chave

Palavras-Chave (Português)

Acknowledgments

I would like to thank the Academy, bla bla bla..

Anyone who has never made a mistake has never tried anything new.

Albert Einstein

Contents

List of Figures

List of Tables

Notation

Latin Letters

A Cross-sectional area [m^2]. 4

a Total surface area per unit length [m]. 4

C_D Drag coefficient [ ]. 4

Greek Letters

γ Adiabatic index $\frac{c_p}{c_V}$ [$J\,kg^{-1}\,K^{-1}$/($J\,kg^{-1}\,K^{-1}$)]. 4

Subscripts

$_p$ Related to the pump. 4

$_v$ Vapour. 4

Rates and Ratios

Eu Euler number $\Delta P/(\rho_v u_v^2)$, where ΔP is the pressure difference between the absorber and the evaporator. 4

$\dot{m}$ Mass flow rate [$kg\,s^{-1}$]. 4

$u_{v/s}$ Slip ratio $\frac{u_v}{u_s}$. 4

1

Introduction

Contents

1.1 Motivation

Motivation Section.

1.2 State of The Art

State of The Art Section.

1.2.1 Dummy Subsection A

State of Art Subsection A

1.2.2 Dummy Subsection B

State of Art Subsection B

1.3 Original Contributions

Contributions Section.

1.4 Thesis Outline

Outline Section.

2

Static Application Security Testing: Scope and Opportunities

Contents

Catalog of academic and non-academic SAST tools.

2.1 Section A

2.1.1 Subsection A

This would be a citation [**?**].

The Coefficient of Performance (COP) defines the performance of the machine.

Heat Pump's performance is given by the Heat Pump Coefficient of Performance (COP_{HP}), a COP for heat pumps.

2.2 Section B

2.2.1 Subsection A

3

SAST Testing and Validation

Contents

Present the chapter content.

3.1 Introduction

Delays in deploying patches to known software security vulnerabilities have been the cause of major cybersecurity attacks [**?** **?** **?** **?** **?**]. A practical example with major financial and reputation losses was the Equifax breach [**?**]: a failure to patch a 2-month-old critical bug in Apache Struts, which led to a sensitive data breach that impacted more than 143 million US consumers [**?**]. Timely patch management (i.e., the fast distribution and deployment of security fixes to users [**?** **?** **?** **?**]) is one of the most effective and widely recognized strategies for protecting software systems against cyberattacks [**?** **?**]. Yet, one important challenge still prevails, the *lack of efficient patch triage systems* to identify and prioritize security patches [**?** **?** **?** **?**]: current processes are largely manual (time-consuming) and prone to ignore important bug fixes such as the one behind Equifax [**?**]. An ideal and effective patch management process should include an effective audit system to identify patches [**?**]. A recent study showed that 56.7% of the commit messages attached to security patches are documented poorly and hinder triage systems tasks (e.g., detection, prioritization, and assessment) [**?**].

Problem: Poor quality security commit messages hinder patch triage systems. Previous work focused on using patch metadata (e.g., commit message) and code changes to explore automated patch detection [**?** **?** **?** **?**], concluded that software vulnerability management (SVM) techniques could not rely purely on metadata to detect software vulnerabilities due to the often inaccurate and incomplete data [**?**]. In fact, only 38% of commit messages used to "silently" patch software vulnerabilities in the past included security-related words [**?**]. Silent fixes are performed when the vendor patches a vulnerability without mentioning its existence anywhere (e.g., release notes, commit message, and more) [**?**]. This practice naturally leads to less informative patch documentation and hinders triage systems awareness and effectiveness. While this practice is usually used to protect vendors and systems, it also limits the general knowledge pool of people who actually understand the vulnerability and know how to exploit it, which leaves users and defenders unprotected and unaware. According to the CERT Coordinated Vulnerability Disclosure (CVD) guide, silent patches should be avoided since knowing the existence of vulnerabilities and their patches is often the key driver to effective patch deployment [**?**].

Motivation: Security commit messages (i.e., the commit messages attached to the code changes used to patch software vulnerabilities) can be cryptic [**?**], hindering triage tasks. Table **??** shows examples of security commit messages (col. Message) used to document patches to known software vulnerabilities (col. VulnID) and the patch commit key (col. SHA). Message 1 (".") is an example of a cryptic message as it does not provide any information. From messages 2 and 3, it is possible to infer that a defect in code is being fixed, but nothing more than that. Message 4 is an internet

	Commit Message	VulnID	SHA
1	.	CVE-2019-13568	ac80033
2	Minor patch.	OSV-2020-2108	a8bf10e
3	Code refactoring.	GHSA-4fc4-4p5g-6w89	d158413
4	<scratchsig><script>location='https:www.youtube.comwatch?v=dQw4w9WgXcQ';<script><scratchsig>	CVE-2020-15179	4160a39
5	Fixed security issues with passwords entered via a prompt	CVE-2022-35928	6876185

Table 3.1: Examples of commit messages used to patch known software vulnerabilities.

meme called “Rickrolling”[1], usually used to prank people with the "Never Gonna Give You Up" song. Although funny, the message does not provide, again, any relevant information regarding the vulnerability and respective patch. Most of the cases presented would not be detected by triage systems leaving users unaware of the new patches and exposed to vulnerabilities. Message 5 provides a brief description of the patch and some relevant information about it (e.g., it fixes a “security issue” related to “passwords”). The respective code changes fix a potential buffer overrun vulnerability resulting from reading user-provided passwords and confirmations via command-line prompts. The patch fixes an “Improper Authentication” weakness (CWE-287) with a severity (CVSS) score of 8.4 in 10—which is not explicit in the message. This extra information could have helped automated or even manual patch triage systems to prioritize this patch since it has high severity.

Study. While some argue that security commit messages and patch release notes should be minimalist, others argue that the details are crucial to ensure triage systems effectiveness [**?** **?**], create trust amongst users [**?**], and enable fast patch management [**?** **?**]. Therefore, we investigated the current status of security commit messages and answered the following questions: (1) what information is included in commit messages of public security patches; (2) are security engineers following best practices to produce security commit messages; and, (3) is the security community open to a new standard for security commit messages. In our work, we empirically analyzed 11036 security commit messages by extracting key information (security words, vulnerability ID, weakness ID, severity, and more) with a customized named entity recognition (NER) tool.

Results. We found that 61.2% of security commit messages include security-related words but lack key information such as the vulnerability ID, weakness ID, and severity. We were unable to extract any information from 8% of the messages because (1) they were poorly documented, (2) their vocabulary was non-security related, or (3) they had misspelled words. Overall, security engineers poorly follow best practices to write general commit messages indicating that a set of new best practices for this task are needed.

Solution. To address the challenges identified from our empirical analysis, we developed a structured and comprehensive convention called SECOM for writing security commit messages. We noticed security engineers sometimes follow standards to write general commit messages in security commit messages (e.g., Conventional Commits [**?**]). Therefore, we designed the standard on top of best

[1] “Rickrolling” meme details at https://knowyourmeme.com/memes/rickroll

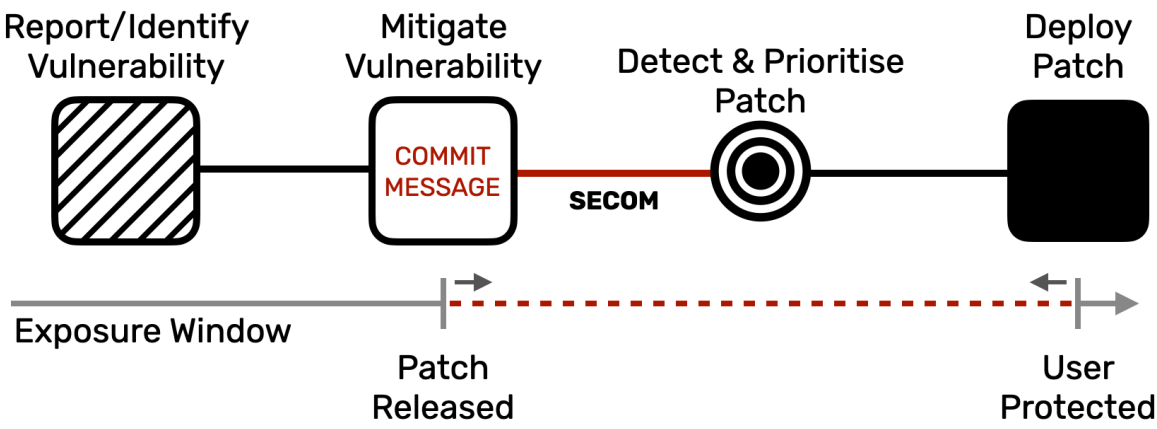


Figure 3.1: Problem and Solution illustration.

practices for writing generic commit messages to facilitate adoption. SECOM was designed according to (1) the results of the empirical analysis of historical commit messages and (2) validation of SECOM with the open-source security community. The standard was created with the aim of making the task of automated reasoning tools easier for patch detection (by adding a clear indicator for vulnerability fixes) and prioritization (by considering severity score and weakness type).

Impact. Figure **??** shows where SECOM can help improve patch management processes: well-documented commit messages can be detected by triage systems and shorten the time between the patch is released and deployed. From a research perspective, detecting and assessing vulnerabilities continues to be a main challenge in vulnerability prediction due to the scarcity and poor quality of curated data [**?**]. SECOM can be an important tool in softening this issue since it has the potential to improve the quality of data for future dataset creation. More than 2k security commit messages have been produced with the SECOM convention[2] so far. We are aware of the challenges that come from too much transparency, and we want to guide the community to make proper and careful use of this new standard. Therefore, in this paper, we also provide guidelines on using SECOM carefully.

Contribution. In summary, our contributions are the following: (1) an empirical analysis of security commit messages and best practices application; (2) a standard for security commit messages, called SECOM; and, (3) guidelines on how to write better security commit messages and apply the convention carefully.

3.2 Background

This section provides background information on the Named Entity Recognition (NER) theory, the approach used to extract key information from security commit messages.

3.2.1 Named Entity Recognition

[2] https://github.com/JLLeitschuh/security-research/issues/8

Figure 3.2: Named Entity Recognition (NER) application example for a security commit message.

Named Entity Recognition (NER) is a form of Natural Language Processing (NLP)—also referred to as entity chunking, extraction, or identification. It is the task of identifying and extracting key information, called *entities*, from unstructured data (in this case, text) [**? ? ?**].

An *entity* can be any word or bag of words that refer to the same *entity category*. For instance, different names of companies "Netflix", "Google" or "Apple" are entities that belong to the *Company* category. NER requires the design of specific entity categories and the respective entity values, which relies on good domain knowledge. NER has been applied to different domains in the past, such as biomedical sciences [**?**], pharmacology [**?**], and even security.

3.2.2 Named Entity Recognition in Security

Previous work used NER to extract product names and versions from vulnerability reports [**?**]. In our empirical analysis, we designed a group of category entities that are usually found in security commit messages and used a customized NER tool for security to extract key information (or entities) from commit messages used to patch software vulnerabilities. Figure **??** shows an example of how we applied NER to security commit messages. Our tool extracted 4 different entities ("fixed", "security", "issue", "passwords") for 3 different category entities ("ACTION", "SECWORD", "FLAW"). Only relevant words are extracted and tagged with their respective category entities, which is crucial for a better understanding of the meaningful information provided in security commit messages.

3.3 Research Questions

We use a mixed-method approach to answer three research questions. The first two research questions, RQ1 and RQ2, are answered by a cross-sectional empirical study of existing security commit messages. The third research question, RQ3, is answered through a survey study.

RQ1. What information is included in the commit messages of public security patches?

Previous work has shown that SVM techniques can not rely purely on commit messages due to poor data quality [**?**]. A study on vulnerabilities patched "silently" showed that only 38% of commit messages used to fix vulnerabilities included security-related words [**?**]. But some previous approaches have managed to find some relevant natural language information in commit messages and perform security patch detection [**? ? ? ? ?**]. The question that remains is what information is being mentioned in the commit messages of security patches.

RQ2. Do security engineers follow best practices to write security commit messages? One way to produce quality commit messages is by following best practices or guidelines [**?** **?** **?** **?**]. However, researchers found that 44% of commit messages need improvement [**?**]. Our research revealed a lack of standards for writing security commit messages. Instead, we found standards and guidelines for generic commit messages [**?** **?** **?**]. Therefore, we explored if available guidelines or standards were being used and how they could be leveraged to create more structured and complete commit messages for security patches.

RQ3. How open is the security community to a new standard for security commit messages?. Standards are usually seen as a burden or with resistance. However, in order to fix challenges, we need to create them. Therefore, we validated our convention with the open-source security community (Open Source Security Foundation).

3.4 Study of Existing Security Commit Messages

This section describes an exploratory, cross-sectional empirical study of security commit messages collected from security patches for known vulnerabilities for answering RQ1 and RQ2.

3.4.1 Dataset Collection and Preprocessing

This section presents the steps taken to create a dataset of security commit messages. All the tools used to collect and preprocess the data are available in our replication package. Our dataset considers data released until the 12th of August, 2022. We collected security commit messages for our exploratory analysis by following the steps described below.

3.4.1.A Vulnerability Metadata Collection from Public Vulnerability Databases

Public vulnerability databases, such as the National Vulnerability Database (NVD) [**?**], and the Open-Source Vulnerability (OSV) database [**?**], integrate documentation (or reports) for thousands of known vulnerabilities. Our dump of the OSV database includes a total of 30091 vulnerability reports for open-source vulnerabilities from different ecosystems: 28.6% of the vulnerabilities were reported by GitHub Advisories, 25.7% by Linux, 11.1% by PyPI, 8.5% by NPM, 7.8% by OSS-Fuzz, and, the remaining 18.3% by the rest of the sources (e.g., Maven, RubyGems, Go, and more). OSV only includes reports of vulnerabilities published after 2005 (inclusive) and vulnerabilities reported by a restricted group of ecosystems. NVD includes reports of known vulnerabilities published since 1999 and has no restrictions regarding the ecosystem (as far as we know). Therefore, we also considered the NVD database in our study. For NVD, we collected a total of 181614 vulnerability reports. In total, we collected 211705 vulnerability reports from both databases.

3.4.1.B Collection and Preprocessing of References to Security Patches

Each vulnerability report for both data sources includes a section referencing the fix (or patch), when available. An example of such a section can be found in [**?**].

Table 3.2: Entity category names, rationale, and entity examples.

Type	Category	Rationale	Entity Examples	R
SEC	SECWORD	Security-relevant words are usually used to describe the vulnerability and respective fix (we used a large set of security-relevant words collected in previous work [**? ?**]).	ldap injection, crlf injection, improper validation, command injection, cross-site scripting, sanitize, bypass	17
	VULNID	Vulnerability IDs are used to identify vulnerabilities for different ecosystems in commit messages: CVE, GHSA, OSV, PyPI, etc. We crafted rules for the different IDs patterns.	GHSA-269q-hmxg-m83q, CVE-2016-2512, CVE-2015-8309, GHSA-9x4c-63pf-525f, OSV-2016-1	9
	CWEID	Vulnerabilities usually belong to a weakness type. One common taxonomy used to classify security weaknesses is the Common Weakness Enumeration (CWE) one. Therefore, we crafted rules to detect CWE IDs.	CWE-119, CWE-20, CWE-79, CWE-189	2
	SEVERITY	Vulnerabilities usually have a severity assigned.	low, medium, high, critical	4
	DETECTION	Vulnerabilities are detected manually or using specific tools.	Manual, CodeQL, Coverity, OSS-Fuzz, libfuzzer	8
COM	SHA	Commit hashes that reference older versions where the vulnerability was introduced (OSV Schema [**?**]).	f8d773084564, 228a782c2dd0	2
	ACTION	A commit usually implies an action, in the case of security, fixing a vulnerability (corrective maintenance).	fix, patch, change, add, remove, found, protect, update, optimize, mitigate	18
	FLAW	Fixing a security vulnerability usually implies fixing a flaw.	defect, weakness, flaw, fault, bug, issue	10
	ISSUE	The GitHub issue/pull request number is sometimes referenced in the message and can provide more information on the vulnerability.	#2, #13245	1
	EMAIL	Contact e-mails of reviewers and authors usually appear after tags such as 'Reported-by' and are important to know who to contact.	`johndoe123@gmail.com`, `catlover@yahooo.com`, `adventuretime@hotmail.com`, `supercool@outlook.com`[1]	1
	URL	Links to reports, blog posts, and bug-trackers references provide more information about the vulnerability.	`https://www.htbridge.ch/advisory/multiple_vulnerabilities_in_mantisbt.html`	1
	VERSION	Software versions are commonly referenced in commit messages.	3.1.0, v3.2, v2.6.28, 1.6.3, 2.1.395	4

Type SEC: Security specific entity categories; **Type COM**: Commit specific entity categories.
[1] **Artificial** e-mails generated automatically with ChatGPT for compliance with General Data Protection Regulation (GDPR).

Collection: To get the commits involved in patching the security vulnerability, we filtered out all the vulnerability reports without references to commit links. We discovered that only 10010 out of 30091 OSV reports and 9953 out of 181614 NVD reports have references to commits (i.e., only 9% of the vulnerability reports in those databases have fixes available). In addition, we observed that commits are usually available through GitHub, Bitbucket, SVN, and other services. In this study, we only focus on vulnerability reports that include commits for fixes (i.e, security patches) available on GitHub, which accounts for over 80% of the commits extracted from vulnerability reports. In total, we found references to GitHub fixes in 8670 NVD reports and 9576 OSV reports.

Preprocessing: In order to collect the commit message of these commits, we used the GitHub API, which requires knowing the *owner* of the repository that integrates the commit; the *name of the repository*; and, the *version* (or, `SHA` key) that included the vulnerability. However, sometimes due to the lack of precise information, we could not determine the data required to get the commit message (e.g., when the commit link had master instead of a specific `SHA` key). Thus, we could not ensure that the current version on master was the version where the vulnerability had been detected. Therefore, we removed all the vulnerability reports exhibiting this issue, which resulted in a total of 8405 security patches for NVD (3% of data points) and 9466 security patches for OSV (1% of data points).

Merging and Cleaning: Both data sources were merged after normalization into a dataset of 17871 security patches while keeping the vulnerability reports metadata. Many vulnerabilities are reported in both NVD and OSV. Therefore, we found duplicates between both sources using different heuristics: 1) duplicated entries for security patches but with missing values for vulnerability score in one of the sources (18% of data points); 2) OSV reports contain a field called "aliases" which is a list

of IDs of the same vulnerability in other databases. Therefore, we removed all the NVD entries (19% of data points) whose IDs where already in the aliases of OSV reports—OSV data was prioritized since previous research has shown that NVD has documentation problems and OSV is making an effort to fix those problems; 3) vulnerabilities fixed with the same patch, usually vulnerabilities that affect different codebases and therefore result in different vulnerability reports were also removed (13% of data points). After removing the different types of duplicates, we end up with a dataset of 10254 security patches.

3.4.1.C Collection and Preprocessing of Security Commit Messages

Vulnerabilities can be fixed with one commit (single-commit patch) or multiple commits (multi-commit patch)—88.6% (9083) of the patches are single-commit patches while the other 11.4% (1170) are multi-commit patches. From 10254 security patches, we extracted a total of 11809 security commits. GitHub metadata (including the commit message) was collected using the GitHub API. The commit messages were preprocessed in different ways: **(1)** A total of 334 commits (the equivalent to 160 vulnerability reports) were *no longer available* at the metadata collection time. Therefore, they were removed from the dataset. **(2)** We found *duplicated commit messages* resulting from vulnerability reports with references to the vulnerability fix but deployed in different branches. One example is the GHSA-273r-mgr4-v34f[3], which references a commit per branch where the vulnerability was fixed. In these cases, since the commit messages are the same, we only kept one of the commits. Therefore, an extra 270 commits were removed from the dataset—which left us with 11205 security commit messages. **(3)** As in previous work [**?**], we searched for the same *non-human generated message patterns* except when the original commit message was somehow attached to the commit message under analysis. For instance, in cases with the pattern "<original commit message> (cherry picked from commit <commit>)", the original commit message is attached at the beginning, and, in the "merge pull request ... <original commit message>" pattern, the original commit message is attached at the end. Therefore, we only remove non-human written messages that do not include any text generated by humans. One example is the `GHSA-3m93-m4q6-mc6v`[4] advisory, which only references the cherry-picked commit. In addition to the patterns mentioned in [**?**], we also removed commit messages with pull request merges from dependabot and merge pull requests without any human text such as "merge pull request from ghsa-g4hm-6vfr-q3wg". In summary, we found and removed a total of 126 automated commit messages and kept 11079 security commit messages. **(4)** We noticed some of the commit messages were not written in English. We ran `langdetect`[5] to infer the message's language. The model detected 1311 (11.8%) security commit messages as non-English. The tool can perform inaccurate predictions when evaluating too short or too ambiguous text. Therefore, we manually inspected the non-English messages to make sure we would not remove English and valid messages. After manual validation, we removed an extra 43 non-English messages such as "导

[3] https://github.com/advisories/GHSA-273r-mgr4-v34f

[4] https://github.com/advisories/GHSA-3m93-m4q6-mc6v

[5] `langdetect`is an algorithm to infer the natural text language. It supports 55 different text languages. Available at `https://github.com/Mimino666/langdetect`.

入mysql db时报错" or "用户头像上传格式限制".

Results. We successfully collected a total of 11036 security commit messages (corresponding to 9943 security patches). This dataset includes security commit messages used to document patches for 278 different security weaknesses.

3.4.2 Data Extraction (R1)

This section explains the methodology used to extract key information from commit messages and answer our first research question.

3.4.2.A Defining Domain-Specific Entities and Categories

We used NER (see Section **??**) to extract key information from security commit messages. Firstly, we designed different entity categories: (1) security-specific or Type SEC, i.e., groups of words or bags-of-words that are common in security commit messages and security vulnerability reports (e.g., SECWORD, VULNID, CWEID, and more); and (2) commit-specific or Type COM (e.g., SHA, ACTION, FLAW, ISSUE, and more), i.e., groups of words or bags-of-words that are common in general commit messages. Table **??** describes the different entity categories, the reason why each of them was considered (*rationale*), entity examples, and the number of rules used to extract data each entity category—which can be fully inspected in our replication package or briefly in Table **??**.

3.4.2.B Named Entity Extraction Pipeline

To extract the entities for each category, we used a Python library called Spacy[6]—which provides end-to-end pipelines for several natural language processing tasks (e.g., NER). We built our own customized NER pipeline for security (see Figure **??**). ***Tokenization.*** Our pipeline takes as input a security commit message that is tokenized (or split into meaningful segments, called *tokens*) using Spacy's tokenizer[7]. ***Part-Of-Speech Tagging.*** Then, the tokens are tagged using the Part-Of-Speech (POS) tagger[8], a pre-trained pipeline component to predict part-of-speech tags[9] such as verb, noun, adjective, adverb, and so on. Pre-trained pipeline components to predict POS tags are language-dependent. Since we focus on English text, we used a pre-trained model for English (`en_core_web_lg`[10]) to extract the POS tags. After collecting the tokens and, respective, POS tags, the pipeline applies a customized set of rules for security commit messages. Spacy's entity ruler[11] enables the customization of entity recognition in text.

3.4.2.C Rules

Fixing security vulnerabilities usually involves an "ACTION" (one of our category entities, Table **??**). Actions can be represented in natural language with verbs such as "fix", "update", "patch",

[6]https://spacy.io/

[7]Spacy's tokenizer documentation: `https://spacy.io/api/tokenizer`. More details on how it works here: `https://spacy.io/usage/linguistic-features#tokenization`.

[8]Spacy's Part-Of-Speech (POS) tagger documentation: `https://spacy.io/api/tagger`.

[9]Universal POS tags available at `https://universaldependencies.org/u/pos/`

[10]`https://spacy.io/models/en#en_core_web_lg`

[11]Spacy's Entity Ruler documentation: `https://spacy.io/usage/rule-based-matching#entityruler`.

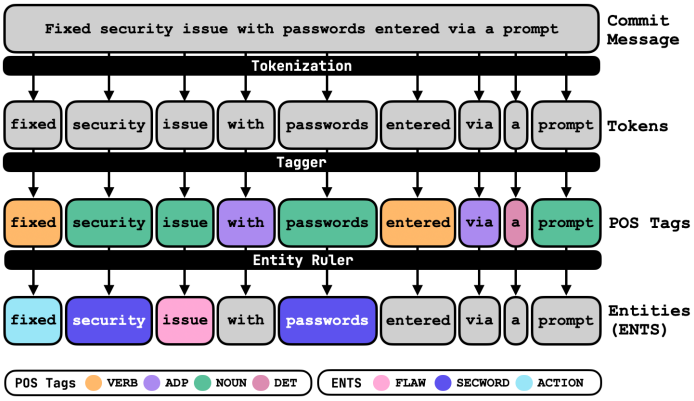


Figure 3.3: Extraction Pipeline

Table 3.3: NER Rule examples. All rules are available in our replication package (`entity_ruler/patterns.jsonl`).

ID	Label	Rule
R1	ACTION	`{"label":"ACTION","pattern":[{"LOWER": {"REGEX":"fix.*"},"POS":"VERB"}]}`
R2	VULNID	`{"label":"VULNID","pattern":[{"LOWER": "osv"},{"IS_PUNCT":true,"OP":"?"}, {"LOWER":{"REGEX":"\\d{4}"}}, {"IS_PUNCT":true,"OP":"?"}, {"LIKE_NUM":true}],"id":"OSV"}`

"mitigate" and more. Therefore, we created several rules that search for different verbal forms of these words. Table **??** shows two examples of rules used by our entity ruler. The first rule extracts all the different verbal forms of the word "fix" (e.g., "fix", "fixing", "fixed", "fixes"), i.e., it extracts all variations of the token "fix" when its POS tag is a `VERB`.

The second example illustrates the extraction of different vulnerability IDs. With the growth of the open-source security community, different ecosystems (e.g., PyPI, NPM, Ruby Gems, and more) are starting to report vulnerabilities with their own IDs that follow different structures than the usual CVE ID, `CVE-\d-\d{4,7}`. Therefore, we implemented rules to extract the different vulnerability IDs (VULNID). One example is R2, a rule to extract vulnerability IDs of type (or rule id) OSV for vulnerabilities detected with Google's OSS-Fuzzer (e.g., OSV-2023- 27[12]).

In addition, we created a total of 1719 rules—from words or bags of words collected in previous work [**?** **?**]. These rules extract security-related words (`SECWORD`). Many rules were improved after manual inspection of commit messages, where (1) we detected erroneous extraction when looking at the entities (e.g., due to broken tokenization of GitHub Advisory IDs, we had to create several different rules based on different tokenization results); or, (2) the tool did not extract any entity. This process led us to augment the list of category entities and find a set of anti-patterns for security commit messages—which is described in detail on Table **??**.

3.4.3 Best Practices Analysis (RQ2)

One way to produce quality commit messages is by following best practices or guidelines [**?** **?** **?** **?**]. In this part of the study, we explored if some of the available guidelines or standards were

[12]https://osv.dev/vulnerability/OSV-2023-27

Table 3.4: Best Practices to Write Generic Commit Messages

ID	Best Practice	Standard
C1	The header should be prefixed with a type.	[**?**]
C2	The message should have a one-line header/-subject.	[**? ? ?**]
C3	The message should have a body.	[**? ?**]
C4	The message should mention the contact of the author (signed-off-by and authored-by).	[**? ?**]
C5	The message should mention the contact of the reviewer (reviewed-by).	[**? ?**]
C6	The message should mention references to issues or pull requests.	[**?**]
C7	The message should include references to bug trackers.	[**?**]

already under usage and how they could be leveraged to create more structured and complete commit messages for security patches. We found six main suggestions to produce good commit messages from four sources of guidelines [**? ? ? ?**]: (1) conventional commits suggest adding a type as a prefix to the subject/header such as `fix:` or `feat:` [**?**]; (2) the header should explain the commit in one line and meaningful [**? ?**] in the capitalized form, no period in the end and the imperative form [**?**]; (3) the body should explain the problem (what), its impact (why) and the fix (how) [**? ?**]; (4) the message should include references to bug-trackers, issues or pull requests (when GitHub is used to manage defects) [**?**]; (5) contacts of the reviewers and reporters [**?**]; and finally, (6) keep commits atomic, i.e., one task per commit [**?**].

We translated the suggestions mentioned before into seven different compliance checkers, which we used to assess if security engineers are following best practices. Table **??** shows the different compliance checkers and their sources.

3.5 SECOM: A Convention for Security Commit Messages and its Validation

To address the problems mentioned in the previous section, we propose SECOM, a convention for writing security commit messages; and, validate its usage with the security community. The new convention helps us answer the research question RQ3.

3.5.1 Design of the Convention

In our empirical study, we observed that although security commit messages do not follow best practices in general, there is a small percentage of people using them—i.e., best practices are being used by some security engineers even if only by a small percentage (Section **??**). To facilitate adoption, we created `SECOM` on top of a well-known group of guidelines for writing good commit messages [**? ? ? ?**]. Table **??** lists and describes the different fields and the reason why each field was considered (*rationale*). For instance, the “type” field was considered because, according to the Conventional Commits Specification, the header/subject should start with a type (4.10% of security commit messages include a type at the beginning of the header). In addition, the Google OSV team

suggested that type should be indeed considered and proposed a new word for security patches, “vuln-fix”.

The structure and set of fields included in the convention were inferred from (1) our empirical analysis of security commit messages collected from security patches available in vulnerability databases such as NVD and OSV; (2) feedback collected alongside the Open Source Security Foundation community; and, (3) previous work on best practices for commit messages [**? ? ? ? ?**].

The convention (Listing **??**) consists of five main sections: **header**, prefixed with the type `vuln-fix`, a simple description of the vulnerability and its identifier (when available); **body**, describes the vulnerability (what), its impact (why) and the patch to fix the vulnerability (how); **metadata**, such as type of weakness (CWE-ID), severity, CVSS, detection methods, report link, and version of the software where the vulnerability was introduced; **contacts**, the names and e-mail contacts of the **reporters** and **reviewers**; and, finally, **references** to bug trackers. The different sections should be separated with a new line.

Listing 3.1: SECOM Convention

```
<type>: <header/subject> (<Vuln-ID>)

<body>
# (what) describe the vulnerability
# (why) describe its impact
# (how) describe the patch/fix

[For Each Weakness in Weaknesses:]
Weakness: <Weakness Name or CWE-ID>
Severity: <Low, Medium, High, Critical>
CVSS: <Severity Numerical Repr. (0-10)>
Detection: <Detection Method>
Report: <Report Link>
Introduced in: <Commit Hash>
[End]

Reported-by: <Name> (<Contact>)
Reviewed-by: <Name> (<Contact>)
Co-authored-by: <Name> (<Contact>)
Signed-off-by: <Name> (<Contact>)

Bug-tracker: <Bug-tracker Link>
OR
Resolves: <Issue/PR No.>
See also: <Issue/PR No.>
```

The convention considers security patches should be atomic [**?**], i.e., two weaknesses can be patched by the same fix, but if it requires more than one fix, then it should be a different commit.

3.5.2 Compliance Checklist

Table **??** provides a checklist (or set of rules) to produce better security commit messages. For each section of the convention, practitioners will find the fields that should be added to the security commit message and questions they should ask when filling them out. We find all the fields important; however, for the sake of prioritization when time is short, we selected some of them as

Field	Description	Rationale
type	Usage of `vuln-fix` at the beginning of the header/subject to specify the fix is related to a vulnerability.	A **type** should be assigned to each commit [**?**]—which will make the identification of vulnerability fixes easier. The `vuln-fix` value was proposed by the Google OSV team during the feedback collection **(F)** phase. In addition, 4.10% of commits follow the conventional commits convention "<type>(scope):".
Header/Subject	It should be approximately 50 chars (max 72 chars), capitalized with no period in the end and in the imperative form.	According to the common best practices for commit messages, it is important to summarize the purpose of the commit in one line [**?** **?**]. In our best practices analysis, we observed that 100% of commit messages had a header, but only 38.85% had security-related words and represented an action.
Vuln-ID	When available, e.g., CVE, OSV, GHSA, and other formats.	Adding the vulnerability ID to the header/subject can help to localize the commit responsible for patching the vulnerability faster using features like reflog or shortlog. Only 12.1% of commit messages included mentions of the vulnerability ID, but 4 out of the 7 participants in **(F)** phase found including the vulnerability ID in the message important.
Body	Describe the vulnerability (what), its impact (why), and the patch to fix the vulnerability (how) in approximately 75 words (25 words per point).	The body is the most important part of the commit message since it provides space to add details on the problem, impact, and solution [**?**]. In our empirical analysis, we observed that 59.91% commit messages have a body. However, only 4031 out of those 6875 cases included security-related words or had meaningful information.
Weakness	Common Weakness Enumeration ID or name.	The weakness ID provides information on which type of vulnerability can exist in the software. Software patch management teams may proceed differently according to the type of weakness. However, only 0.2% of messages included this type of information.
Severity	Severity of the issue (Low, Medium, High, Critical).	Severity can motivate software users to perform patch management faster (in case of critical vulnerabilities) [**?**]. Again, only 1.1% of commit messages mentioned severity levels.
CVSS	Numerical (0-10) representation of the severity of a security vulnerability (Common Vulnerability Scoring System).	CVSS allows users to make better sense of the vulnerability severity and can motivate software users to perform patch management faster [**?**]. This field was proposed by a security engineer at OpenSSF that mentioned that sometimes is possible to calculate the score by following the CVSS questionnaire.
Detection	Detection method (Tool, Manual, etc).	It can be interesting to help future researchers with replication. 4 out of the 7 participants in the **(F)** phase sees value in adding this field (Table **??**, RQ2).
Report	Link for vulnerability report, which can back up the lack of information provided in commit messages.	It usually provides more information on the vulnerability exploit or proof-of-concept. We observed that 3 out of the 7 participants would like to see links to reports, **(F)** phase (RQ1).
Introduced in	Commit hash from the commit that introduced the vulnerability.	Suggested by a survey participant of the (**F**) phase and used in the OSV Schema [**?**]. In addition, we found SHA keys in 1467 commit messages.
Signed-off by	Name and contact of the person that reported the issue.	To provide credit to the person that found the problem and ask for more details when necessary. However, only 8.4% of commit messages were signed off by the respective authors.
Reviewed-by	Name and contact of the person that reviewed and closed the issue.	Reviewers are usually the internal developers or senior developers that review and approve the issues. Only 3.33% of messages have the reviewers' contact.
Bug-tracker	Link to the issue in an external bug-tracker or `Resolves...` `See also:` when GitHub is used to manage issues.	Important to document and discuss the problem, its impact and know which people were involved. In our empirical analysis, we extracted URLs from a total of 929 commits.

Table 3.5: Fields description and rationale.

Header	type	Did you set the type of the commit as "vuln-fix" at the beginning of the header?	**M**
	header/subject	Did you summarize the patch changes?	**M**
	header/subject	Did you summarize the patch changes within ∼50 chars?	**O**
	Vuln-ID	Is there a vulnerability ID available? Did you include it between parentheses at the end of the header?	**M**
Body	what	Did you describe the vulnerability or problem in the first sentence of the body?	**M**
	why	Did you describe the impact of the vulnerability in the second sentence of the body?	**M**
	how	Did you describe how the vulnerability was fixed in the third sentence?	**M**
	*	Did you describe the what, why, and how within ∼75 words (∼25 words per section)?	**O**
Metadata	Weakness	Can this vulnerability be classified with a type? If so, add it to the metadata section.	**M**
	Severity	Can infer severity (Low, Medium, High, Critical) for this vulnerability? If so, add it to the metadata section.	**M**
	CVSS	Can you calculate the numerical representation of the severity through the Common Vulnerability Scoring System calculator (https://www.first.org/cvss/calculator/3.0)?	**M**
	Detection	How did you find this vulnerability? (e.g., Tool, Manual, etc)	**O**
	Report	Is there a link for the vulnerability report available? If so, include it.	**O**
	Introduced in	Include the commit hash from the commit where the vulnerability was introduced.	**O**
Contacts	Reviewed-by	Include the name and/or contact of the person that reviewed and accepted the patch.	**O**
	Signed-off-by	Include the name and/or contact of the person that authored the patch.	**M**
Bug-Tracker	External	Include the link to the issues or pull requests in the external bug-tracker.	**O**
	GitHub	Include the links for the issues and pull-requests related to the patch (`Resolves.. See also:`).	**O**

Table 3.6: SECOM Compliance Checklist. [**M**-Mandatory; **O**-Optional; ***-All fields in the section.]

mandatory—which are the ones we think to be most important to detect (type, Vuln-ID), prioritize (Weakness, Severity, CVSS) and understand the message (header, what, why, how). However, practitioners should always try to make security commit messages as detailed as possible. The compliance validation with SECOM's structure (Listing **??**) and rules (Table **??**) is currently automated with a tool. In the future, we plan to explore how to generate suggestions to produce better security commit messages or even entire ones to soften the burden of a new standard in maintenance teams.

3.5.3 SECOM's Application Example

SECOM is meant to structure and help write more informative and meaningful security commit messages. In this section, we show an application example of SECOM to improve the commit message used to document the patch to CVE-2012-0036[13], a potential data injection via a crafted URL.

Listing 3.2: Original commit message to fix CVE-2012-0036

```
URL sanitize: reject URLs containing bad data
Protocols (IMAP, POP3 and SMTP) that use the path
part of a URL in a decoded manner now use the new
Curl_urldecode() function to reject URLs with
embedded control codes (anything that is or
decodes to a byte value less than 32).

URLs containing such codes could easily otherwise
be used to do harm and allow users to do
unintended actions with otherwise innocent tools
and applications. Like for example using a URL
like pop3://pop3.example.com/1%0d%0aDELE%201
when the app wants a URL to get a mail and
instead this would delete one.

This flaw is considered a security
vulnerability: CVE-2012-0036

Security advisory at:
http://curl.haxx.se/docs/adv_20120124.html

Reported by: Dan Fandrich
```

Listing 3.3: Commit message to fix CVE-2012-0036 (after SECOM's application)

[13] https://nvd.nist.gov/vuln/detail/CVE-2012-0036

```
vuln-fix: Sanitize URLs to reject malicious data
(CVE-2012-0036)

Protocols (IMAP, POP3 and SMTP) that use the path
part of a URL in a decoded manner now use the new
Curl_urldecode() function to reject URLs with
embedded control codes (anything that is or
decodes to a byte value less than 32).
URLs containing such codes could easily otherwise
be used to do harm and allow users to do
unintended actions with otherwise innocent tools
and applications.
Like for example using a URL like
pop3://pop3.example.com/1%0d%0aDELE%201 when the
app wants a URL to get a mail and instead this
would delete one.

Weakness: CWE-89
Severity: High
Detection: Manual
Report: https://curl.se/docs/CVE-2012-0036.html

Reported-by: Dan Fandrich
Signed-off-by: Daniel Stenberg (daniel@haxx.se)

Resolves: #17940
See also: #17937
```

Listing **??** shows the original commit message used to document the code changes to patch the CVE-2012-0036. Listing **??** shows an example of the same commit message, but if the message had followed the SECOM standard convention and rules. As seen, the header includes a type, a short message, and the vulnerability ID, followed by the body where a description of the problem and fix is provided, and later all the information regarding metadata, contacts, and bug trackers are also provided. In the original message, details such as weakness and severity were not provided, which would not allow triage systems to prioritize this patch properly.

3.5.4 Standard Validation (RQ3)

This section describes how we collected feedback (F) from the security community about the proposed convention (SECOM). We prepared a small survey to collect feedback from the open-source security community.

The survey did not collect any personal data, and we informed that to our participants at the beginning of the form. SECOM was presented and discussed in detail in two working groups of the Open-Source Security Foundation (OpenSSF). OpenSSF is an open community where security engineers from all over the industry are involved (e.g., Google, Linux Foundation, Intel, RedHat, and more) and on a mission to make open-source security better [**?**]. The convention was validated by experienced security engineers involved in the best practices and vulnerability disclosure projects. At the end of our discussion, we asked the present security engineers to provide their feedback on the convention by answering a few questions (see Table **??**). These questions were designed to validate the different fields we proposed (Q1 and Q2) and to understand the community's opinion on our new solution (Q3-Q5). For instance, we wanted to understand which field the community finds more important. One thing we observed in our manual analysis of commit messages was the mention of the process of detection of the vulnerability—sometimes, the tool used to detect the vulnerability was mentioned in the message. Since we did not have any strong data analysis to backup this decision, we asked the potential users what they thought about it.

3.6 Findings

In this section, we present the main conclusions of our empirical analysis of security commit messages and the validation of SECOM.

RQ1: What information is included in the commit messages of public security patches?

We ran our extraction pipeline (Figure **??**) over a total of 11036 security commit messages. Table **??** shows the number of entities extracted per entity category (#Entities), the number of commits where entities of each type were found (#Commits), and the respective percentage of commits (%Commits). We divided the entities into two main sets: security-specific and commit-specific. The tool was able to extract key information for both types: 38.5% of the extracted entities are security-specific, and the remaining 61.5% are commit-specific.

Commit-specific data (COM). The tool extracted entities for all the 7 different entity categories classified as commit-specific. A total of 29841 entities were extracted as commit-specific information. A commit usually implies an *action* or *change*. In the case of security, it should imply fixing a vulnerability (corrective maintenance). Yet, only 10364 entities for verbs (ACTION) were extracted from 6409 (58.1%) commit messages. Fixing a vulnerability also means fixing a flaw. Our tool extracted mentions of flaws (FLAW) in 2843 (25.8%) commit messages. Mentions of issues (ISSUE) and URLs (URL) can be a good source of external documentation and information, but security engineers rarely mention that information in their commit messages: only 2805 contained references to issues, and 929 commit messages contained URLs.

Security-specific data (SEC). The tool extracted entities for all the 5 different entity categories classified as commit-specific. A total of 16126 security-related words (SECWORD) was collected from

No.	Question	Answer
Q1	"The following pieces of information commonly appear in commit messages of security-related patches. Which do you find important to include in a security commit message?"	Vuln-ID, Severity, CVSS, Weakness Type, Report Link
Q2	"Security vulnerabilities are usually detected by different means (e.g., manually, static analysis, dynamic analysis, penetration testing, etc.). Do you see value in reporting this information in a security commit message?"	Yes, No, Unsure
Q3	"Would you use this or a similar convention as standard practice in your own work or advocate its use in your team?"	Yes, No, Unsure
Q4	"If you answered "Other" or "Unsure to any of the questions, please explain briefly below."	Open
Q5	"Please enter any other comments or suggestions below."	Open

Table 3.7: Survey questions used to validate SECOM.

Table 3.8: Extraction Results

Category	#Entities	#Commits	%Commits
SECWORD	16126	6749	61.2%
ACTION	10364	6409	58.1%
EMAIL	4738	2086	18.9%
SHA	4943	1467	13.3%
FLAW	4402	2843	25.8%
ISSUE	3561	2805	25.4%
URL	1175	929	8.4%
VULNID	1799	1330	12.1%
VERSION	658	571	5.2%
DETECTION	629	374	3.4%
SEVERITY	142	118	1.1%
CWEID	25	23	0.2%
Total	48562	10168	92.1%

6749 out of 11036 (61.2%) commit messages. Vulnerability IDs (VULNID) were only extracted for 1330 commit messages, while all patches that integrate our dataset, fix a vulnerability with a known ID. Vulnerability IDs easily map security commits to official reports, which usually provide more details on the vulnerability, product affected, severity, and more. Therefore, they are important to add to commit messages. It seems that security engineers rarely mention weakness IDs (CWEID)—only 25 entities were extracted from 23 commits messages. We suspect that we can often extract the weakness type by looking at the set of entities extracted for the SECWORD category. For instance, in "fixed xss vulnerability bug by oncellhtmldata" (commit message from security patch to the GHSA-hf4q-52x6-4p57 vulnerability), we could infer a CWE-79 based on the "xss" entity extracted in this message. Severity can be important for security patch management systems to know which security patches to prioritize. If a patch to a critical vulnerability is released, it should be installed as soon as possible. However, severity entities (SEVERITY) are rarely included in security commit messages—only extracted from 118 (1.1%) security commit messages.

Finding 1. Security engineers use security-related words in 61.2% of the security commit messages used to patch software vulnerabilities.

Finding 2. Vulnerability IDs, Weakness IDs and Severity are rarely mentioned in security commit messages—although important for manual and automated detection and prioritization.

Our tool extracted a total of 48562 entities from 10168 out of the 11036 (92.1%) security commit messages under analysis. For the remaining 868 of the commit messages, we performed a manual validation of the reason behind of null extraction:

Poorly-Written. We found several types of poorly-written messages reported in previous work [**?**]. For instance, we found 51 messages containing only one token ("Single-word"). Some examples are "update", "...", ":arrow_up:", "Refactor". Poorly-written messages usually contain one line without clear information regarding the commit's purpose (e.g., "applied updates", "backend media". We found a total of 760 messages that lacked meaningful information.

Non-Security Related. Dense message with no clear relationship with security. One example is "Support progressive event for dc. This implements the progressive api event fo rthe dc image. This is currently only supported for vardct without extra channels: for modular and extra channels it's only supported if squeeze is used, and it may not correctly work with flush yet in that case."—the commit

message of OSV-2021-1606 vulnerability. We found a total of 97 commit messages that were hard to relate with security fixes.

Misspelling Issues. Messages with misspelled english words that would be detected if well written. One example is the message: “sanitzing user input ”. However, this kind of issues reflects a very small percentage of the problem.

Finding 3. No extraction of entities was performed from 8% of security commit messages mainly due to poorly written messages, misspelling issues and no clear connection with security.

RQ2. Do security engineers follow best practices to write security commit messages?

We applied 7 compliance checkers to the commit messages to evaluate if security engineers are using generic best practices to write commit messages since no standard for security exists.

C1. One way of easily marking a commit message for an automated solution is with the usage of a prefix in the header (as the Conventional Commit Convention proposes [**?**]). In security, this practice was only used in 453 out of 11036 (4.10%) of commits follow the conventional commits convention “<type>(scope):” using prefixes such as “patch” or “fix”.

C2. Headers are important because they are ultimately used to make a quick searches of relevant commits through the git reflog feature. Therefore, it is important that all commit messages have a meaningful and clear header. All the security commit messages (100.00%) have a one line subject/-header. But only 4288 out of 11036 (38.85%) headers have security-related words (SECWORD) and reflect an action (ACTION).

C3. Previous work has shown that a good commit message needs to clearly mention the problem (what), its impact (why) and how it will be fixed (how) [**?**]. However, only 6875 out of 11475 (59.91%) commit messages have a body. Security engineers do not use often security related-words (or vocab) in their body messages since only 4031 out of 11036 (36.53%) of body messages have SECWORDS.

C4. Crediting the authors of the fixes by signing-off by commits is a very well known practice in security. However, only 925 out of 11036 (8.4%) commit messages were signed-off by the authors.

C5. Reviewers are usually the internal developers or seniors developers that review and approve the issues. Only 368 out of 11036 (3.33%) of messages have the reviewers contact.

C6. Issues are good sources of documentation since they sometimes provide details on the discussion of the problem and potential solution. However, only 2805 out of 11036 (25.42%) commit messages have references to issues.

C7. Same as references to issues, links to bug trackers are also good to mention since they usually provide useful documentation on the problem, fix, severity, and more. But again, only 196 out of 11036 (1.78%) of commits have references to bug trackers.

Finding 4. Security engineers, do not follow best practices to write security commit messages in general. Even when it seems they are, we concluded that key information is missing—which indicates we need best practices for writing better security commit messages.

RQ3. How open is the security community to a new standard for security commit messages?

Feedback received from the security community suggests that they see value in SECOM and would like to see it evolve into a standard practice—6 out of the 9 participants responded “Yes” to Q3 (“Would you use this or a similar convention as standard practice in your own work or advocate its use in your team?”), the remaining three participants answered “Unsure”. None of the participants answered “No”. 2 out of 9 participants said they did not find valuable to mention how the vulnerability was detected. However, 6 of the 9 participants answered “yes”. Therefore, we considered the “Detection” field in the convention. The vulnerability ID, CWE ID, and severity are the fields that participants found more important to include in the commit message. The least important fields are the CVSS and report link.

Finding 5. The security community sees value in SECOM and aims to adopt it as a standard practice in the future.

The participants that were unsure of SECOM’s adoption were mainly concerned with the current practices: “Folks won’t be practicing this style daily, and for drive-by contributions, git messages are likely to be highly ad hoc and idiosyncratic.”. However, we argue that with good automated tools, we can help developers produce better commit messages for security. Writing more structured and informative commit messages is important since, ultimately, it can improve the detection and assessment capabilities of patch triage systems and enable fast patch management.

3.7 Implications and Considerations

In this section, we describe the implications of our findings, ethical considerations, and how SECOM should be used.

3.7.1 Dealing With Patch Transparency

Transparency is a double-edged sword. It can be a source of trust for consumers [**?**], but it also creates vulnerability [**?**], and, the need for security is often used to avoid transparency because of the risks that come from it. The reality is that non-transparent processes lead to abuse and make timely patch management a challenge (or even impossible). The CERT Coordinated Vulnerability Disclosure (CVD) guide suggests avoiding silent patches since it hinders public awareness of fixes to software vulnerabilities. This, then, leads to a lack of understanding and trust from developers and to poor triage systems (automated tools are incapable of reasoning about data and detect patches if no key information is provided).

The SECOM convention and best practices purposed in this paper are meant to help vendors to produce better documentation and boost the patch management phase—responsible for deploying the new changes to the users. We are aware that being too much transparent can make users vulnerable to cyberattacks, but we argue that providing better documentation will help automated tools be more effective and fast for software security patch management. So, OSS users and companies

can benefit from an automated solution by being aware of the fixes as soon as they are developed, which is usually one week earlier than the disclosure [**?**].

Experienced security engineers should determine when and not when to provide details. If development systems are private, then no major risks are attached to providing a detailed security commit message (in principle, we do not account for internal attackers). However, when a critical vulnerability is to be patched in open-source software of wide use, then security engineers should carefully decide which details to include in the commit message. They should provide enough information to users understand its criticality but not enough to attackers leverage.

One thing we do not advocate is the documentation of exploits since this would clearly benefit malicious attackers. That kind of information should be released later, after providing time for the users to deploy the patch.

3.7.2 The Burden of a New Standard

Applying new best practices is usually taken as a burden by the software and security communities, but it is also how we prevent and solve current challenges. As we mentioned before, well-structured and complete documentation is crucial to create trust between different parties and faster patch management—to decrease the user's exposure window. Therefore, we, as a team, and part of the software engineering community, are working on automated solutions for compliance validation and the generation of commit messages to reduce the bottleneck that our standard could introduce. `SECOM` can also be leveraged to guide other solutions to generate commit messages [**?**]. It is also important to notice that the open-source security community sees value in SECOM and aims to adopt it as a standard practice in the future. In fact, some security researchers are already using it and considering it as standard best practice [**?**], and more than 2k commit messages were produced following our solution.

3.7.3 Need for Better Patch Documentation

Previous work has shown that the software patch detection problem could not be solved only with security commit messages (or other types of metadata) [**?** **?**]—which we agree. The solution proposed is to focus on code information instead. One study proposed a mixed solution [**?**]. However, we think that focusing on code information will not fix the issue soon due to the complexity of the patterns under analysis (software vulnerability), the approaches gaps (e.g., machine learning, deep learning [**?**]), the diversity of weaknesses and many more. Instead, we can make an extra effort to provide better documentation and not only improve automated solutions but also build trust between vendors and users—which we argue will ultimately build a safer environment.

3.8 Threats to Validity

This section discusses the study's potential threats to validity.

Internal Validity: In the survey study, our sample of participants is small, which may not reflect the entire population. However, we ensured that all the answers come from a reliable and expert source (OpenSSF). In addition, we only considered participants from the open-source community (i.e., private software developers may not be represented).

External Validity: We did not check for all the rules described in the standards considered. Therefore, our conclusion may not reflect the entire ground truth. But, we did check the most important rules, the ones related to having a type, header, and body in the commit message which improves the commit messages considerably.

3.9 Related Work

This section summarizes previous research work in the fields of Software Security Patch Management, Software Security Patch Detection and Commit Message Analysis.

3.9.1 Software Security Patch Management

Software security patch management is the process of identifying, acquiring, testing, installing, and verifying security patches for software products and systems [**?**]. Security patches are considered the most effective strategy to mitigate software vulnerabilities [**? ? ?**]. They are prioritized over non-security patches as they aim to protect users from cyberattacks [**?**]. Yet, one important challenge still prevails, the *lack of efficient patch triage systems* to identify and prioritize security patches [**? ? ? ?**]: current processes are largely manual (time-consuming) and prone to ignore important bug fixes such as the one behind Equifax [**?**]. Figure **??** shows part of the patch management process: *1)* vendors release a patch to mitigate a vulnerability previously identified or reported; *2)* automated or manual systems identify and prioritize the patch and trigger patch deployment; *3)* the patch is deployed successfully into the users' machines. In this paper, we propose a solution that can improve the understanding and identification of security patches for automated and manual triage systems, which can consequently decrease the time users are exposed to vulnerabilities. Silent fixes, i.e., software vulnerabilities patched without any release note, hinder security patch management since they leave users unaware of the new patches and their criticality [**?**]—knowledge about the fix is key for successful patch deployment. Previous work has shown that 56.7% of the security commit messages are documented poorly [**?**], which hinders the different triage systems tasks. This paper presents a solution to make security commit messages more informative and enable the effectiveness of patch management processes.

3.9.2 Software Security Patch Detection

Version control systems are used to maintain a record of code changes and manage access to codebase development artifacts [**?**]. As with any type of code changes, security patches are pushed to codebases through commits. Each commit contains changes to source code and a message explaining what changes are made and why [**? ?**]. Over the past years, many approaches have

emerged to detect security patches through metadata (e.g., commit logs, commit metadata, and associated reports) and code changes. Here, we only mention approaches considering commit logs (or messages). *Reis et al.* detected security patches for 16 types of weaknesses in commit messages by applying different regular expressions per each kind of weakness [**?**]. *Zhou and Asankhaya* leveraged regular expressions to identify features for predicting security-relevant commits in commit logs, commit metadata, and associated bug reports. Features are vectorized and learned by word2vec [**?**]. *Sabetta and Bezzi* extended previous work by considering code changes as well [**?**]—which led to considerable performance improvements. *Sawadogo et al.* proposed SSPCatcher, a co-Training-based approach to catch security patches as part of an automatic monitoring service of code repositories [**?**]. Detection is based on the result of two Support Vector Machine classifiers: 1) one trained with commit logs; and, 2) one trained with code features. In order to identify a new security patch, both classifiers have to be in agreement. In this study, we do not propose a new model or approach to detect security patches. Instead, we empirically analyze security commit messages and propose a standard that can improve the effectiveness of the different triage systems tasks (e.g., detection, assessment, and prioritization).

3.9.3 Security Commit Message Analysis and Standards

Previous work has shown that only 38% of security commit messages used to "silently" patch software vulnerabilities in the past included security-related words [**?**]. In this work, we analyzed the commit messages attached to all security patches to known software vulnerabilities instead of only focusing on silent fixes. As mentioned before, we were unable to find a convention or standard for security commit messages. Instead, we only found guidelines and specifications for general commit messages. The preliminary results of this study were published before [**?**]. This paper is an extension and detailed report on how we analyzed the data, the results, and the design of our solution.

3.10 Conclusions and Future Work

In this study, we observed that security commit messages integrate some relevant information for automated and manual triage systems but not enough to map them to bug reports or know which patches to prioritize. We also observed that no standards exist for security and that security engineers usually do not follow best practices. This paper presents a solution for the lack of structure, and key information in the commit messages of security fixes. we built SECOM, a convention for security commit messages, based on our empirical analysis findings and the open-source security community feedback and approval.

Although part of the security community worries about the attacks that potentially will come from transparency, there seems to be space in the security community for a new standard that is already helping security engineers produce better security commit messages. However, it should be used carefully and consider the software vulnerability impact and time response of the maintenance team. In the future, we plan to release automated solutions for compliance validation and message gen-

eration to reduce the burden that a new best practice can introduce in the patching and disclosure life-cycle.

4

Infrastructure-as-Code Scripts Application

Contents

This chapter studies and improves the precision and recall of IaC tools through practitioners feedback and Generative Artificial Intelligence.

4.1 Introduction

4.2 Leveraging Practitoners Feedback to Improve Precision

Software configuration management and deployment tools like Puppet[1], Ansible[2], and Chef[3] became popular amongst software development warehouses [**?** **?**]. The adoption of these tools has increased with the growing movement to run software in cloud servers. These tools help infrastructure teams increase productivity by automating various configuration tasks (e.g., server setup). In short, these tools describe the environment configuration in a set of provisioning scripts that can be versioned and reused. They enable configuration consistency between different environments and can reduce the time required to provision and scale the infrastructure. The process of managing and provisioning infrastructure through configuration scripts is called Infrastructure-as-Code (IaC). IaC tools take a script as input and create an infrastructure that typically runs in a virtual environment as output.

As with any piece of code, IaC scripts are also prone to defects such as security vulnerabilities [**?**]. For example, in 2020, Palo Alto Network researchers reported the discovery of over 199K vulnerable IaC templates [**?**]. Specifically, 42% of AWS CloudFormation templates, 22% of Terraform templates, and 9% of Google Kubernetes YAML files were vulnerable. In addition, researchers found more than 67k *potential* security smells in IaC scripts implemented in Ansible, Chef, and Puppet [**?**] through an ad-hoc tool created to show the presence of a new set of anti-patterns for security in the IaC domain. These reports highlight the importance of tools to prevent vulnerabilities from reaching production and shift security left in the development pipeline.

Figure **??** shows an example of an *Admin by default* weakness (CWE-250[4]), a potential vulnerability in a Puppet manifest in a module of the `PuppetLabs`.[5] The vulnerability manifests when the developer configures a user as “admin” or “root” for an infrastructure component. In this example, the `$grafana_user` is set as “admin” for the different services (Puppetserver, Puppetdb, Postgresql, Filesync) used by Grafana[6]. Therefore, any service can be prone to a privilege escalation attack. In IaC, all the infrastructure components are configured through scripts, including the access credentials. Specifying default users as administrative gives privileges to users that only an administrator should have. Admin accounts can be exploited to access sensitive data and execute unauthorized code/commands. Infrastructure engineers should avoid setting admin passwords and usernames to user accounts that do not need the privileges. Detecting these issues automatically essential to make

[1] `https://puppet.com/`

[2] `https://www.ansible.com/`

[3] `https://www.chef.io/`

[4] CWE-250 details available at `https://cwe.mitre.org/data/definitions/250.html`

[5] Admin by default example available at `https://github.com/puppetlabs/puppet_operational_dashboards/blob/9eb67a407aa44c2f924f67f207edc7032f81f86a/manifests/profile/dashboards.pp#L137` (Accessed 11 de outubro de 2023)

[6] Grafana is an open-source software application for data exploration and visualization. More information available at `https://grafana.com/`

```
['Puppetserver','Puppetdb','Postgresql','Filesync'].each |$service|{
  grafana_dashboard { "${service} Performance":
    grafana_user     => 'admin',
    grafana_password => $grafana_password.unwrap,
    grafana_url      => $grafana_url,
    content          => file("../${service}_performance.json"),
    require          => Service['grafana-server'],
  }}
```

Figure 4.1: Simplified example of an *Admin by default*.

infrastructure engineers aware of possible security problems and to help companies build more robust infrastructures.

Problem: Puppet IaC Security Linters are not reliable yet! In 2019, Rahman et al. showed that IaC scripts—just like any piece of code—are not immune to security vulnerabilities [**? ?**]. They focused on Puppet configuration files and listed seven anti-patterns that could lead to security vulnerabilities. The work led to the development of SLIC, a linter to detect those defects in Puppet scripts. Linters are often imprecise tools [**? ? ? ? ? ?**]. Therefore, motivated by the report of very high accuracy (i.e., precision and recall) from their paper (Table IV [**?**]), we decided to conduct a reproduction study of SLIC based on a different and larger set of projects. We asked students (co-authors of this paper) and developers to analyze the warnings that the tool reports. To validate the students observations of low precision, we reported a sample of the warnings of the tool to maintainers of 86 open-source projects. From the 228 issues created, we obtained responses to 51 issues where only 33 issues were discussed or clear. Results showed that the tool performs differently in a new set of projects, particularly when validated by the software owners. The precision observed was smaller than the one reported in SLIC's original paper (28% instead of 99%) which indicates that security IaC linters for Puppet are not reliable yet due to the high false positive rates.

Like many linters, SLIC uses simple rules to detect issues. Essentially, it searches for string patterns in the values of tokens (many times) regardless of their type (e.g., variable, string, etc.) and the relationship between them. For instance, the "Usage of Crypto. Algorithms" checker (CWE-326[7]) searches for any token whose value includes `sha1` or `md5`. Both are built-in Puppet functions and SLIC fails to consider the context of usage of these algorithms, i.e., these functions are called in Puppet manifests to encrypt data (e.g., $encrypt\_key = md5(\texttt{key})$). Therefore, SLIC incorrectly detects `md5checksum = '07bd73571b7028b73fc8ed19bc85226d'` as a CWE-326. This simplicity creates much noise for developers. In this preliminary study, we observed that the rules for the current IaC security anti-patterns must be better designed to be safely adopted by the industry and avoid productivity disruption.

Solution: Our preliminary study revealed that (1) there is a need to improve the precision of IaC security linters for Puppet, and that (2) security tools can be iteratively improved and extended by

[7]CWE-326 details are available at `https://cwe.mitre.org/data/definitions/326.html`

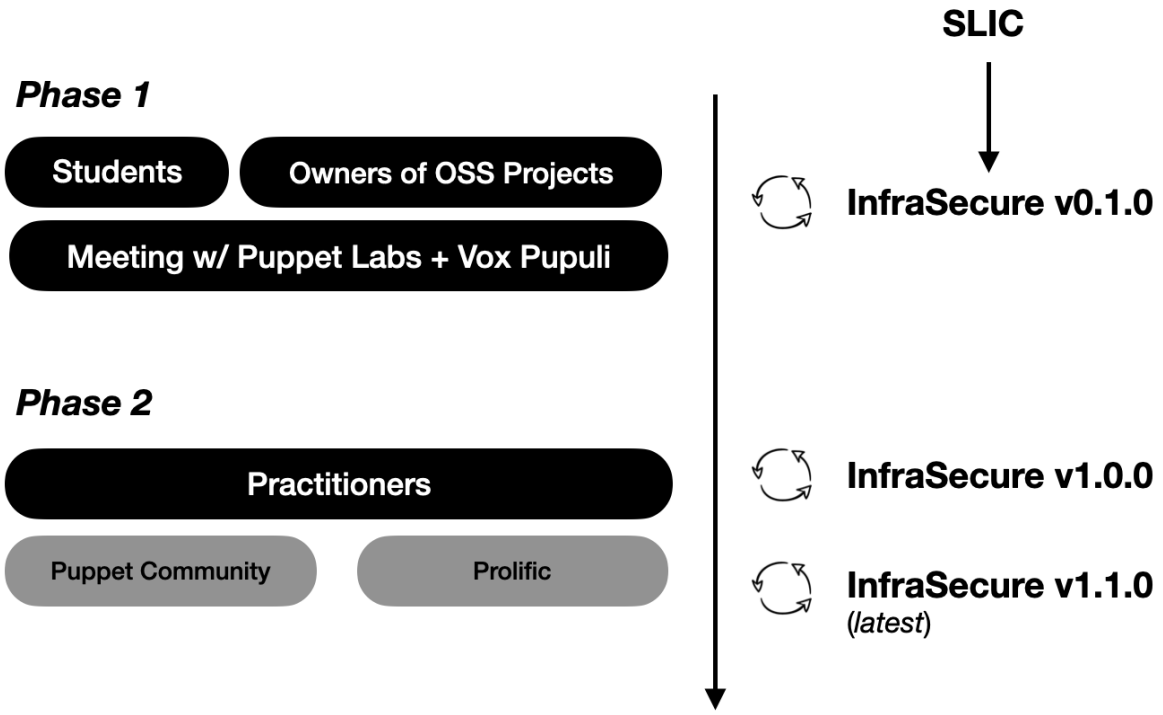


Figure 4.2: Timeline of feedback collection.

incorporating feedback from the developer community as suggested in previous work [**?**]. This paper reports on the process we followed to iteratively and incrementally improve the precision of an IaC linter according to user feedback. For example, the experiments described above ignited discussions with members of the development and security teams of Puppetlabs[8], as well as one project manager from Vox Pupuli[9]. The feedback collected from the team, OSS maintainers and the Puppet community led to the creation of a new tool, which we dubbed as INFRASECURE. Later, we leveraged the expertise of practitioners experienced in IaC tools or security to iteratively and incrementally improve the new tool.

Figure **??** shows the timeline of feedback collection followed to design and improve INFRASECURE. To sum up, we bootstrapped the design of INFRASECURE with rules obtained from the revision of SLIC's ruleset, according to the feedback of the research team and owners of OSS projects (*phase 1* in Figure **??**); and, incrementally evolved the linter according to the recommendations of practitioners (*phase 2* in Figure **??**). We improved 7 rules of the SLIC ruleset and added 3 new rules that were either recommended by practitioners (e.g. weak password); or relevant for the infrastructure domain (e.g. homograph attacks[10] and malicious dependencies).

Main Results: This paper performs the following contributions:

⋆ **Study.** A replication study of `SLIC`'s precision, including a preliminary study conducted with two researchers (co-authors of this paper) and a study with several GitHub scripts validated by project maintainers;

[8]GitHub PuppetLabs organization website: `https://github.com/puppetlabs`

[9]Vox Pupuli is the organization responsible for maintaining modules and tools for the Puppet community: `https://voxpupuli.org/`

[10]Apple Domain Attack (2017): `https://www.xudongz.com/blog/2017/idn-phishing/`

Table 4.1: Examples of security smells per weakness.

CWE	Example
CWE-798	$username = “mariadb”
CWE-259	$password = “!TQ23Rg”
CWE-321	$key = “A67ANBD7”
CWE-319	$req = “http://www.domain.org/secret”
CWE-546	#https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=538392
CWE-326	password => md5($debian_password)
CWE-284	$bind_host = “0.0.0.0”
CWE-258	$rabbitmq_pwd = “”
CWE-250	$user = “admin”
CWE-521	pwd => “12345”
CWE-1007	$source = "http://deb.debian.org/debian"
CWE-829	$postgresql_version = 8.4

⋆ **InfraSecure.** A new linter adjusting 7 rules of the original SLIC ruleset and adding 3 new rules with a final precision of 83%.

⋆ **Dataset.** A dataset of IaC scripts with more than 1000 warnings classified as false positives and true positives that researchers can use to evaluate other security linters;

Take-away message: The takeaway message of this paper is that it is feasible to tune security linters to produce acceptable precision for important classes of warnings (confirming the findings reported in a study at Google [**?**]) and that involving practitioners in discussions is an effective way to guide the improvement of those linters.

Replication Package: All the scripts and data used in this study (including feedback obtained from the maintainers and practitioners) are available at: `https://figshare.com/s/6b6a769b1393eae0774c`.

4.3 Background

This section provides background information on Puppet and discusses security weaknesses in IaC scripts.

4.3.1 Puppet

Developers no longer deal with systems composed of a single machine and database. Instead, system administrators must manage multiple diverse operating systems, databases, and virtual machines. Perhaps most importantly, they must ensure their configurations are consistent at any given time. Configuration management technologies have been around for over a decade—Puppet was founded in 2005. Puppet is a solution that helps IT infrastructure management through code by supporting software deployment, packages, and configuration. In Puppet, programs are written using Puppet’s declarative language. They are called manifests. More information about the language can be found elsewhere[11].

[11] More about Puppet: https://puppet.com/docs/puppet/7/puppet_overview.html

4.3.2 Security Weaknesses

This section describes in more detail the potential weaknesses in Puppet scripts. Table **??** illustrates examples of each weakness.

Hard-coded secrets (CWE-798): This warning refers to the practice of including sensitive information such as passwords or cryptographic keys in code files. Table **??** shows a hard-coded password example. Rahman et al. considered 3 kinds of data as sensitive: usernames, passwords, and cryptographic keys. However, the Common Weakness Enumeration[12] list does not consider solo hard-coded usernames as a threat. Practitioners involved in our validation studies shared the same opinion. Therefore, we argue that hard-coded usernames should be only reported when there is a paired password (CWE-259) or cryptographic key (CWE-321). More discussion on this is provided later in Section **??**.

Use of HTTP without TLS (CWE-319): This warning refers to the practice of using HTTP without the Transport Layer Security (TLS) to transmit sensitive data. This means that attackers can more efficiently exploit the communication channel as the data is transmitted unencrypted, as cleartext[13].

Suspicious comment (CWE-546): This warning refers to comments that suggest the presence of bugs, missing security functionalities, or weaknesses of a system. Details provided in comments about bugs, security functionalities or weaknesses can be crucial for hackers to exploit the infrastructure.

Use of weak cryptography algorithms (CWE-326): This warning refers to using weak crypto algorithms. Attackers can easily crack the encryption schemes and have access to the data [**?**].

Invalid IP address binding (CWE-284): This warning refers to assigning the address 0.0.0.0 to a server. This practice allows connections from any IP address to access that server [**?**]. While mail servers have to listen on 0.0.0.0 to receive mail, database servers should not since it can lead to critical data breaches.

Empty password (CWE-258): This warning refers to using empty strings as passwords, which are easily guessable.

Admin by default (CWE-250): As detailed in the Introduction, this warning refers to defining users with administrative privileges. This can result in security weaknesses since it can disable or bypass security checks performed by the system.[14]

Weak Password (CWE-521): This warning refers to the usage of weak passwords. Weak passwords are easily guessable and can be bypassed to gain access to systems.

Insufficient Visual Distinction of Homoglyphs Presented to User (CWE-1007): This warning refers to malicious actors using homoglyphs that may cause the user to misinterpret a glyph and perform an unintended, insecure action. The homograph attack performed against the apple website[15] is a well-known example of this type of weakness. Table **??** shows an example of a domain where

[12] CWE-798 description is available at `https://cwe.mitre.org/data/definitions/798.html`

[13] This type of issues can lead to man in the middle (MITM) attacks: `https://owasp.org/www-community/attacks/Man-in-the-middle_attack`

[14] Why you should not use an admin account: `https://www.lbmc.com/blog/why-you-should-not-use-an-admin-account/`

[15] Phishing with Unicode domains: `https://www.xudongz.com/blog/2017/idn-phishing/`

Table 4.2: Breakdown of warnings reported by SLIC.

Rule	#	%
Hard-coded secrets	22365	69.9
Use of HTTP without TLS	3757	11.7
Suspicious comments	2780	8.7
Use of Weak Crypto. Algos.	1489	4.7
Invalid IP Address Binding	769	2.4
Empty Password	684	2.1
Admin by default	146	0.5
Total	31990	100

the character "a" could be replaced by the respective homoglyph, as in the apple attack. This warning might be essential to uncover malicious domains implanted by malicious open-source contributors—typosquatting attacks.[16]

Malicious Dependencies (CWE-829): This warning refers to malicious software by nature, i.e., dependencies that integrate known vulnerabilities (CVEs). These are the leading cause of supply chain attacks, and one of the main challenges the security community faces nowadays [**?**].

4.4 Preliminary Study

This section reports on the findings of two studies—involving different sets of participants—to assess the performance of SLIC, a recently-developed linter for Puppet.

4.4.1 Validation with Students

This section reports on a study involving two of this paper's authors to assess the precision of SLIC on an independent benchmark. The study consisted in inspecting 502 warnings reported by the tool. The warnings were validated by one senior PhD student whose research focuses on security and static analysis and one junior PhD student in software engineering with basic security skills.

4.4.1.A Dataset

To build our dataset, we mined GitHub projects containing `Puppet` scripts. We used three different queries to search for repositories: 1) `language:puppet is:public`; 2) `puppet in:readme is:public`; and, 3) `devops is:public`. We discarded results pointing to forked repositories (to avoid duplicates) and discarded results pointing to repositories without any code in `Puppet` scripts. Our crawler found a total of 1419 GitHub repositories and 34574 associated `Puppet` scripts. SLIC scanned these scripts for the seven sins and reported a total of 31990 security warnings involving 9144 `Puppet` scripts (=26.5% of the total) from 1093 repositories (=73.5% of the total). Table **??** shows the breakdown of warnings reported by SLIC. Column "Rule" shows the name (kind) of the warning, column "#" shows the number of warning reports of that kind, and column "%" shows the percentage of the total associated with that number. This table lists the warnings in order of their prevalence.

[16]OpenSSF post on scanning OSS software for malicious behavior: `https://openssf.org/blog/2022/04/28/introducing-package-analysis-scanning-open-source-packages-for-malicious-behavior/`

Table 4.3: Performance of SLIC. (Validation with Students)

SLIC	*proportional*			*uniform*		
Rule	#TP	#FP	Pr.	#TP	#FP	Pr.
Hard-coded secrets	122	52	0.70	26	10	0.72
Use of HTTP without TLS	9	20	0.31	10	26	0.28
Suspicious comments	10	12	0.45	8	28	0.22
Use of Weak Crypto. Algorithms	7	4	0.64	25	11	0.69
Invalid IP Address Binding	6	0	1.00	28	8	0.78
Empty Password	4	2	0.67	21	15	0.58
Admin by default	1	1	0.50	21	15	0.58
Total	159	91	0.64	139	113	0.55

4.4.1.B Methodology

Samples. Given the high number of warnings reported by the tool (31990) and the need for humans to analyze each warning, we sampled a set of reported warnings. We leveraged two popular complementary sampling strategies to that end [**?**]. *Stratified sampling* is a method to draw samples from a set by taking into account the distribution of kinds—in our case, the distribution of kinds of warnings. A *proportional* (stratified) sampler draws samples in a number proportional to the size of the sets associated with each kind whereas an *uniform* (stratified) sampler draws the same number of samples for each kind. Intuitively, a proportional sampler values more the most prevalent kinds of warnings (as to make more accurate measurements on those kinds) whereas an uniform sampler treats every kind equally (as to avoid inaccurate measurements on uncommon kinds). We sampled 250 warnings *proportionally* and 252 (=36*7) warnings *uniformly*. In total, we analyzed 502 warnings, which is a substantial increase when compared to the 58 warnings analyzed in the SLIC's paper [**?**]. **Metric.** We focused on precision to measure the reliability of the reports of the tool. Precision is especially important for security linters. Reporting scores of false warnings can be highly disruptive for a team's productivity, as team members tend to interrupt work to address high-priority tasks [**?**]. Developers are less willing to use linters with low rates of precision because they find them not trustworthy and unreliable [**?**]. **Statistical Tests.** Each one of the 502 warnings was manually inspected by two co-authors. Cases where the authors found disagreement were discussed to reach a consensus. We report on the results of a Cohen's Kappa analysis [**?**] to measure the inter-rater reliability of human decisions.

4.4.1.C Results

Table **??** shows SLIC's results for both sampling strategies: *proportional* and *uniform*. For each sampling strategy, we present the number of true positives (#TP), the number of false positives (#FP), and the Precision. Considering the results for *proportional* sampling, the authors found a total of 159 true positives and 91 false positives. The average precision of SLIC was 0.64 for the proportional set. Considering the results for *uniform* sampling, the authors found a total of 139 true positives and 113 false positives. On average, SLIC's precision was 0.55 for the uniform set.

We ran a Cohen's Kappa analysis to measure the inter-rater reliability of human decisions in classifying the warnings. The kappa coefficient (k) shows the level of agreement between the two co-authors. The analyses yielded k=0.89 and k=0.94 for the *proportional* and the *uniform* sampling

sets, respectively. According to McHugh's interpretation of k [**?**], the reported levels of agreement are strong and almost perfect for the proportional and uniform sampling sets, respectively. For illustration, agreement was reached in 482 out of 502 warnings. The two co-authors discussed the warnings that raised disagreement. Cases where consensus was not reached were replaced by a new one and re-evaluated. Cases where agreement was reached were updated with the final conclusion—inferred from the discussion between both co-authors.

Summary: Results show that the original precision of SLIC drops considerably from 99% (reported in the original work [**?**]) to below 65% when tested in a new set of puppet scripts—which might indicate that SLIC needs to be improved. However, the lack of context on the software under analysis by the co-authors may also be the reason for a lower precision. Therefore, we conducted a new experiment with the owners and maintainers of open-source software, i.e., people with more knowledge and context of the applications.

4.4.2 Validation with Owners of OSS projects

Complementing the preliminary study reported in the previous section, this section reports on the findings of a validation study of SLIC conducted with the maintainers of open source projects. The motivation of this study was to confirm the observations of the previous experiment but now with open source developers, i.e., developers with more context of the software under analysis. GitHub issues were designed to illustrate the security smells (including references to the corresponding CWEs[17]) and to guide the developer towards patching the issue. We followed guidelines for issue reporting from the literature. Issues include code samples, links to more information and we strive to make the report message as brief and objective [**?**]. Figure **??** shows an example of an issue created for a "Hard-coded Secret". The title is "Potential vulnerability in Puppet file: Hard-coded Secret". The issue (1) shows where the potential vulnerability is (code: `cron_user = 'root'`, script: `puppet-apt_mirror/manifests/init.pp` and line: 191); (2) explains the vulnerability and its possible implications (bypass protection mechanism, gain privileges on applications, and access to sensitive data); and (3) makes a recommendation to the developer on how she can fix the vulnerability (by using a vault, in this case). We created different templates of this message for the different warning types.

4.4.2.A Dataset

The dataset used in this study was different from the one from Section **??**. We mined GitHub projects with activity in 2020 (at least one commit) and containing Puppet scripts. We conjectured that focusing on projects with recent activity would increase our chances of obtaining responses. We used two different queries to search for repositories: 1) `language:puppet is:public`; and, 2) `puppet in:readme is:public`. We discarded results pointing to forked repositories (to avoid duplicates), repositories without any activity in 2020 (no commits), and repositories without any code in Puppet per project. Our tool scanned 3740 Puppet scripts from 287 GitHub repositories. In total, 1975 security

[17]Common Weakness Enumeration (CWE) taxonomy available at `http://cwe.mitre.org`.

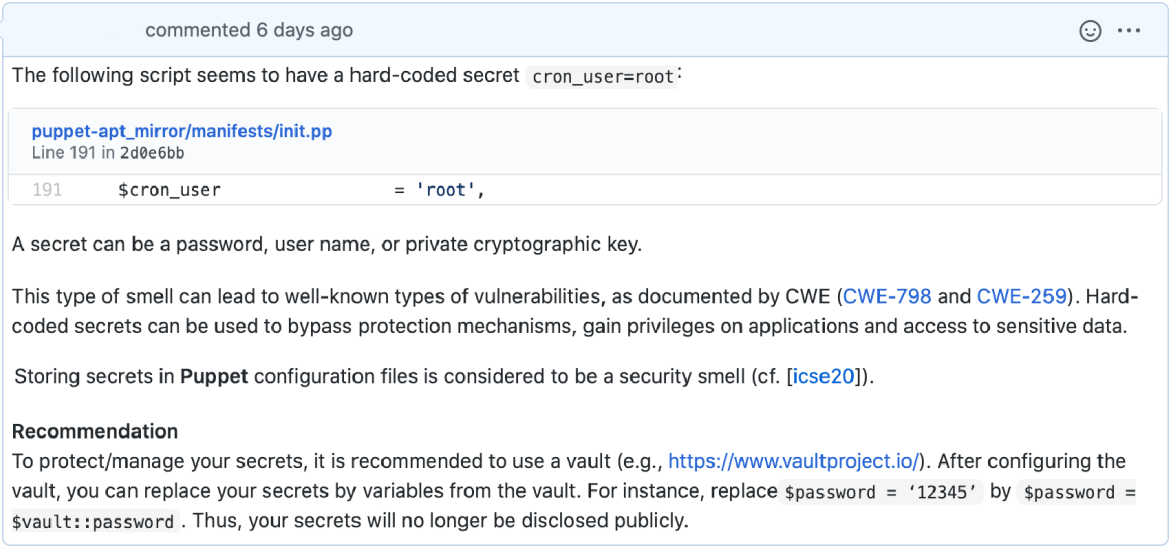


Figure 4.3: Example of issue opened (based on SLIC report).

warnings were detected in 1147 Puppet scripts (=30.7% of the total). We issued alerts to projects with maintainers involved in the slack of the Puppet community. We received 51 answers to the 228 issues we submitted, but only 33 issues were clearly validated by practitioners—which were the ones we considered for our conclusions.

4.4.2.B Methodology

This section presents the methodology followed to submit the issues. **Sample.** A total of 287 GitHub repositories were scanned to this study. We ensured that the repositories had recent activity (at least one commit in 2020) to improve the probability of obtaining responses. The total amount of scripts scanned was 3740 —7 times the sample used in our preliminary study (Section **??**) and 28 times the sample used in Rahman et. al [**?**] to evaluate the SLIC's precision and recall. **Issues.** We reached out to the software owners through the Puppet community slack and submitted issues to projects with maintainers involved in the slack community. All the issues followed a specific template depending on the type of warning (cf. Figure **??**). The issued message not only located and explained the vulnerability but also recommended an example of a patch (i.e., actionable messages to save maintainers time). **Reply Evaluation.** We monitored the discussion threads associated with the issues. For each response obtained, we classified the warnings reported as true positives (TP) or false positives (FP) according to the opinion of the maintainers in their responses. Issues closed by the maintainers without any reply or activity were discarded. Issues closed with unclear responses (e.g., "N/A", ":thumbs_down") were also discarded since they did not provide clarity on the validation of the issue. We only considered the issues where there was some sort of discussion of the issue (e.g., "These todo's shouldn't be there, I agree ... but it's not about defects/weaknesses here. It's just a marker to include more operating systems in the future.") or a clear validation of the issue (e.g., "All false positives", "This is not a secret."). From the answers obtained, two of the co-authors manually inferred the classification of each warning. We use Cohen's Kappa [**?**] to measure the inter-rater reliability of human decisions. **Metrics.** We measured Precision as described in Section **??**.

Table 4.4: Performance of SLIC. (Validation with Owners)

Rule	#TP	#FP	Precision
Hard-coded secrets	77	119	0.39
Use of HTTP without TLS	1	72	0.01
Suspicious comments	3	15	0.17
Use of Weak Crypto. Algos.	0	3	0.00
Invalid IP Address Binding	0	1	0.00
Empty Password	1	5	0.17
Admin by default	1	0	1.00
Total	83	215	0.28

4.4.2.C Results

This section reports results. **Issues.** We reported a total of 1975 warnings in 228 issues (9 warnings per issue, on average). Project owners responded to 51 issues of the 228 issues we submitted, but only 33 issues were clearly discussed or validated—the equivalent to 298 warnings (Table **??**). One issue for an "Empty Password" warning was fixed by one of the maintainers (82c3cb7[18]); one tagged the issue with "waiting for contribution"; another commented asking to perform a pull request. **Reply Evaluation.** We used the following method to determine the warning classification (i.e., false or true positive) from the answers of project owners. We discarded warnings related to issues closed without any response or activity and issues that remained open or without any response by the time of our analysis. After that stage, two of this paper's co-authors reviewed each of the answered issues. Each warning received two votes. Then, we ran a Cohen's Kappa analysis to measure the inter-rater reliability of our choices to assess the confidence in our classification method. The kappa coefficient (k) shows the level of agreement between the two co-authors. The analysis yielded $k = 1.0$, i.e., a total agreement between both co-authors. **Precision.** Table **??** shows the number of true positives (TP) and the number of false positives (FP) per type of warning. In total, SLIC reported 83 true positives and 215 false positives for the 33 issues considered, which resulted in an average precision of 0.28 for SLIC. Note that the samples used for "Use of Weak Crypto. Algorithms", "Invalid IP Address Binding", "Empty Password" and "Admin by default" are relatively small, so results might not reflect the entire reality.

Summary: Results indicate that the precision of SLIC is even lower when evaluated by maintainers—developers with more knowledge and context of the applications—of the software (drops to 28%). These results confirm our initial observations and indicate that better security IaC linters for Puppet are needed.

4.5 INFRASECURE: Puppet Security Linter

The observations described in the previous section motivated the pursuit of a more precise security linter for Puppet scripts. The previous experiments ignited discussions with members of the development and security teams of PuppetLabs, as well as a project manager from Vox Pupuli. We leveraged the feedback obtained from the previous studies and the professional feedback to design a

[18] Fix for "Empty password" issue: `https://github.com/jtopjian/puppet-sshkeys/commit/82c3cb7e78c16cf6517207779f79ab5b2a71b603` (Accessed 11 de outubro de 2023)

new security linter for the Puppet community, which we dubbed as INFRASECURE. More precisely, we created a new linter as a result of *phase 1* (Figure **??**) and incrementally evolved the linter according to the recommendations of professionals (Figure **??**, *phase 2*), improving 7 rules of the SLIC ruleset and adding 3 new rules. The following sections report the new architecture (Section **??**), design choices (Section **??**) and security checkers (Section **??**) that resulted from the feedback collection (Figure **??**):

- *Phase 1*: feedback from the **owners of OSS projects**, **PuppetLabs** and **Voxpupuli Engineers** (as described in Section **??**) that led to the creation of INFRASECURE (v0.1.0);
- *Phase 2*: two cycles of feedback from the **Puppet** and **Prolific** communities (as described in Section **??**) that led to two new releases of INFRASECURE (v1.0.0 and v1.1.0);

4.5.1 Linter Architecture

One recommendation from the PuppetLabs team (*phase 1*) was to implement the set of the rules as plugins to the `puppet-lint` architecture[19], through the `puppet-lint` check API[20]. This API facilitates the integration of new checkers in `puppet-lint`In addition, it allows the user to suppress warnings and disable or enable checkers—which are regarded as important features by the community. All security checks were developed as plugins to `puppet-lint` (Table **??**). These checks are applied to the Abstract Syntax Tree (AST) of a Puppet manifest which is generated by an internal tokenizer[21]. INFRASECURE was implemented in `Ruby` and its `CLI` is available online[22]. The codebase of the linter is available at `https://github.com/TQRG/puppet-lint-infrasecure` and open to future contributions.

4.5.2 Design Choices

This section describes the design choices of our analysis, guided by the distinct cycles of feedback as described in Section **??** and Section **??**; also, illustrated in Figure **??**.

Variable/Attribute Assignments (VASS). From the preliminary analysis performed in Section **??**, we have noticed security-related code smells being detected in logical conditions. For instance, `if has_key($userdata, 'env')` shows a logical condition that was incorrectly flagged as a hard-coded secret issue. Aiming to reduce the number of incorrect predictions, we implemented a rule to search for variable and attribute assignments in Puppet manifests—`isVarAssign(`*`token`*`)` and `isAtrAssign(`*`token`*`)` (cf. Table **??**).

Reasoning about the token value (TOKVAL). Some of the rules did not reason about `token.value`. For hard-coded secrets, the linter only checks if the token value is not empty. While manually validating the samples used in our studies, we found false positives of hard-coded secrets. For instance, `aws_admin_username = downcase($::operatingsystem)` which does not store any actual

[19]Puppet-lint website: `http://puppet-lint.com/`
[20]Puppet-lint check API: `http://puppet-lint.com/developer/api/`
[21]Puppet-lint tokenizer: `http://puppet-lint.com/developer/tokens/`
[22]Gem is available at `https://rubygems.org/gems/puppet-lint-infrasecure`

secret. SLIC flagged this case as a hard-coded secret because the value assigned to the variable `aws_admin_username` is not empty. However, the rule needs to reason not only about the length of the right-hand side of the variable assignment but also about the type of token and value. INFRASECURE locates variable and attribute assignments in the AST and considers that secrets are usually stored in `:STRING` and `:SSTRING` tokens. In addition, we defined a database of known credentials (`isUserDefault(`*`token.value`*`)`)—credentials that are not considered secrets by the community[23]—and, invalid secrets (`invalidSecret(`*`token.value`*`)`) which are consider as non-valid values for hard-coded secrets. The linter ignores all the credentials in this database. Feedback from distinct **owners of OSS Projects** is what drove us to make this decision is presented below:

[User Default]: *"The names of these UNIX accounts are not considered to be secret. They are published openly as part of the PE documentation: https://puppet.com/docs/pe/2019.8/what_gets_installed_and_where.html#user_and_group_accounts_installed"*

[Invalid Secret]: *"This are default users and default as found in every installed fpm package. there is most of the time a wwwrun or a www-data user depending on the system."*

Use of HTTP without TLS is fine sometimes (SAFE). As SLIC, INFRASECURE also flags every single occurrence of *http://*, i.e., it recommends to use TLS by default. For example, the tool flags `apturl => "http://deb.debian.org/debian"`, even though it refers to a credible source. Our definition of credible source is a source that can be trusted. However, different companies can have different opinions regarding the credibility of the same source. That is why this rule is customizable. We observed that this type of issues (CWE-319) are prevalent in Puppet files. Applications often use third-party libraries, which are usually configured in Puppet files, and the links to their sources are not necessarily unsafe. Also, depending on the context of an application, the configuration of localhost servers as HTTP may not be a problem. If no sensitive data is communicated, then there is probably no problem using `http`. INFRASECURE has a configuration file for safe domains, i.e., domains that are cleared to be use `http`. Thereby, infrastructure teams can customize their own configuration files. The feedback provided in Section **??** from two different **practitioners**, which supported this decision, is presented bellow:

[Whitelist]: *"I think it is fine if localhost is used. Otherwise TLS should be mandatory. All the big financial organizations will not use this check because they cannot create internal certs or use letsencrypt."*

[Whitelist]: *"By default, it's unsafe to not use HTTPS. But for internal testing/development it is acceptable to me to not use HTTPS all the time."*

Hard-Coded Secret Division in different checkers (SECR). In Section **??**, we observed that the hard-coded secrets checker produces the most significant number of alerts. For instance, SLIC assumes a secret is a key, password or username. As mentioned previously in Section **??**, the Common Weakness Enumeration list does not consider solo hard-coded usernames as a threat. Practitioners

[23] `https://puppet.com/docs/pe/2019.8/what_gets_installed_and_where.html#user_and_group_accounts_installed`

involved in our validation studies shared the same opinion. We analyzed the distribution of the different types of hard-coded secrets and realized that 48% of the secrets detected were usernames. Therefore, in the final version of our tool, we decided to separate the hard-coded secrets checker into three new checkers (one per type of secret). This way, developers can disable the username checker if they find it noisy. We did not delete the original checker; infrastructure teams can use it if they want to collect all the different types of secrets simultaneously. Feedback provided in Section **??** from a **practitioner** supported this decision:

[Username]: *"The main security issue is having the password hard-coded. About having the user hard-coded, it is possible to allow that as an initial setting that should be changed during the first configuration and, in that case, it is not so much a security issue."*

4.5.3 Rules

INFRASECURE detects 12 different security smells in Puppet manifests. Table **??** presents the AST patterns that are searched in the AST for relevant nodes/sequences of nodes; and, table **??** presents the string patterns used to validate the information in those nodes. Table **??** shows the syntactic pattern matching rules per weakness which leverage the two sets of patterns mentioned before.

Hard-coded secrets (CWE-321, CWE-259, CWE-798): The top of the Table **??** contains 4 different rules for hard-coded secrets: one per secret; and a final one which detects all kinds of secrets at the same time (keys, password and usernames). In addition to the design choices, the rules consider that secret values cannot be placeholders (`!isPlaceholder()`, Table **??**).

Use of HTTP without TLS (CWE-319): One of the main findings of our analysis is that HTTP without TLS is not always problematic. Therefore, we created a configurable whitelist where infrastructure teams can add safe domains. The checker will not raise alerts when in the presence of a safe domain (`inWhitelist()`, Table **??**). INFRASECURE provides a default whitelist with known reliable sources such as `http://deb.debian.org/debian`. However, this default whitelist will be overwritten if the user configures a new one.

Suspicious Comments (CWE-546): This checker was controversial. It was recognized that it would be valuable to alert developers about comments in their code mentioning functionalities and weaknesses that might hint at attackers. However, keywords such as "todo", "later", and "later2" were considered noisy. We changed the list of keywords in response to the complaints and feedback obtained from the developers (`isSuspiciousWord()`, Table **??**).

Usage of Weak Crypto. Algorithms (CWE-326): INFRASECURE searches for *in calls to* functions (`isFunctionCall(), Table` **??**) implementing crypto algorithms such as "md5" and "sha1" in variable and attribute assignments (Table **??**).

Invalid IP Address Binding (CWE-284): We found cases where the invalid IP `0.0.0.0` was in descriptions and commands. For instance, SLIC flags `description => 'Open up postgresql for access to sensu from 0.0.0.0/0'`. IPs follow a dot-decimal notation, i.e., they should not include letters. INFRASECURE uses a less naive regex than the string pattern (`isInvalidIPBind()`, Table **??**).

Table 4.5: INFRASECURE's list of string and AST patterns.

Rule	String Pattern
isAdmin(*t.value*)	`root|admin`
isNonSecret(*t.value*)	`gpg|path|type|buff|zone|mode|tag|header|scheme|length|guid`
isPassword(*t.value*)	`pass(word|_|$)|pwd`
isUser(*t.value*)	`user|usr`
isKey(*t.value*)	`(pvt|priv)+.*(cert|key|rsa|secret|ssl)+`
isPlaceholder(*t.value*)	`${.*}\|($)?.*::.*(::)?`
hasCyrillic(*t.value*)	`^(http(s)?://)?.*\p{Cyrillic}+`
isInvalidIPBind(*t.value*)	`^((http(s)?://)?0.0.0.0(:\d{1,5})?)$`
isSuspiciousWord(*t.value*)	`hack|fixme|ticket|bug|checkme|secur|debug|defect|weak`
isWeakCrypto(*t.value*)	`^(sha1|md5)`
isCheckSum(*t.value*)	`checksum|gpg`
isHTTP(*t.value*)	`^http://.+`
isUserDefault(*t.value*)	`pe-puppet|pe-webserver|pe-puppetdb|pe-postgres|pe-console-services|pe-orchestration-services|pe-ace-server|pe-bolt-server`
invalidSecret(*t.value*)	`undefined|unset|www-data|wwwrun|www|no|yes|[]|undef|true|false|changeit|changeme|none`
isStrongPwd(*t.value*) [a]	StrongPassword::StrengthChecker(*t.value*)
isEmptyPassword(*t.value*)	value == ""
isVersion(*t.value*)	`.*_version`

(a) String patterns are applied to token values.

[a] The `strong_password` ruby gem (`https://rubygems.org/gems/strong_password`) is used to determine if a password is strong or not.

Rule	AST Pattern
isVariable(*t*)	t.type == `:VARIABLE` ∨ t.type == `:NAME`
isString(*t*)	t.type == `:STRING` ∨ t.type == `:SSTRING`
isVarAssign(*t*)	isVariable(*t.prev_code_token*) ∧ t.type == `:EQUALS` ∧ isString(*t.next_code_token*)
isAtrAssign(*t*)	isVariable(*t.prev_code_token*) ∧ t.type == `:FARROW` ∧ isString(*t.next_code_token*)
isResource(*t*)	(t.prev_code_t.type == `:NAME` ∧ t.type == `:LBRACE` ∧ t.next_code_t.type == `:SSTRING`) ∨ (t.prev_code_t.type == `:LBRACE` ∧ t.type == `:SSTRING`)
isFunctionCall(*t*)	(t.type == `:NAME` ∧ t.next_code_token.type == `:LPAREN`) ∨ t.type == `:FUNCTION_NAME`
isComment(*t*)	t.type is in (`:COMMENT`, `:MLCOMMENT`, `:SLASH_COMMENT`)

(b) Patterns applied to the Abstract Syntax Tree (AST).

Table 4.6: Performance of INFRASECURE v0.1.0.

INFRASECURE v0.1.0	*proportional*			*uniform*		
Rule	#TP	#FP	Pr.	#TP	#FP	Pr.
Hard-coded secrets	118	22	0.84	24	4	0.86
Use of HTTP without TLS	8	17	0.32	9	23	0.28
Suspicious comments	5	2	0.71	6	10	0.38
Use of Weak Crypto. Algorithms	5	2	0.71	23	2	0.92
Invalid IP Address Binding	6	0	1.00	28	1	0.97
Empty Password	4	2	0.67	21	15	0.58
Admin by default	1	1	0.50	20	15	0.57
Total	147	46	0.76	131	70	0.65

Empty Password (CWE-258): Empty passwords are located the same way as hard-coded secrets, i.e., by focusing on variable and attribute assignments (Table **??**). The rule `isEmptyPassword()` (Table **??**) verifies if the password is empty.

Admin By Default (CWE-250): These issues are also located by focusing on variable and attribute assignments (Table **??**). The rule `isAdmin()`, table **??**, verifies if the user is "admin" or "root".

Homograph Attacks (CWE-1007): Typosquatting attacks, also known as URL hijacking, is a social engineering attack that purposely uses misspelt domains for malicious purposes; and are the cause of many supply chain attacks [**?**]. This checker is important because malicious actors can use homoglyphs to modify reliable sources for malicious sources (`hasCyrillic()`, Table **??**).

Weak Password (CWE-521): INFRASECURE searches for passwords in the same way it searches for Empty Passwords and Hard-Coded Passwords. The only difference is the password value validation (`isStrongPwd()`, Table **??**) which is performed by an external package (`strong_password`) that implements an adaptation of a PHP algorithm developed by Thomas Hruska [**?**].

Malicious Dependencies (CWE-829): We produced a database of malicious dependencies for Puppet modules by crossing CVEs information and vulnerable products names with third-party libraries that can be configured in Puppet manifests. We used the National Vulnerability Database (NVD) to collect the CVEs and respective vulnerable products—from the list of Known Affected Software Configurations. To get the list of products used by Puppet, we used the Forge API[24]. Our database integrates malicious dependencies for 33 different packages (e.g., `rabbitmq`, `apt`, `cassandra`, `postgresql`, etc) The checker searches for resource configurations (`isResource()`, Table **??**) and verifies if the a configured version of the software integrates our database of malicious dependencies for Puppet (`isMalicious()`, Table **??**).

4.5.4 Proof of Concept: INFRASECURE v0.1.0

As a proof of concept, two of the design choices described in Section **??** were implemented in the first version, INFRASECURE v0.1.0, to ascertain whether precision could be enhanced. In particular, we focused on implementing variable and attribute assignments (VASS) and reasoning about the token value (TOKVAL), to reduce the number of incorrect detections.

In our preliminary analysis with students (see Section **??**, Table **??**), we observed that the pre-

[24]Forge API is available at https://forgeapi.puppet.com/

Table 4.7: INFRASECURE rules to detect security smells.

CWE	Weakness Name	Rule
CWE-321	Hard-coded Key	(isVarAssign(*t*) ∨ isAtrAssign(*t*)) ∧ isKey(*t.prev_code_token*) ∧ isNonSecret(*t.prev_code_token*) ∧ !isPlaceholder(*t.next_code_token*)
CWE-259	Hard-coded Password	(isVarAssign(*t*) ∨ isAtrAssign(*t*)) ∧ isPassword(*t.prev_code_token*) ∧ isNonSecret(*t.prev_code_token*) ∧ !isPlaceholder(*t.next_code_token*) ∧ !isUserDefault(*t.next_code_token*) ∧ !invalidSecret(*t.next_code_token*)
CWE-798	Hard-coded Usernames	(isVarAssign(*t*) ∨ isAtrAssign(*t*)) ∧ isUser(*t.prev_code_token*) ∧ isNonSecret(*t.prev_code_token*) ∧ !isPlaceholder(*t.next_code_token*) ∧ !isUserDefault(*t.next_code_token*) ∧ !invalidSecret(*t.next_code_token*)
CWE-798	Hard-coded Secrets	(isVarAssign(*t*) ∨ isAtrAssign(*t*)) ∧ (isKey(*t.prev_code_token*) ∨ isPassword(*t.prev_code_token*) ∨ isUser(*t.prev_code_token*)) ∧ !isPlaceholder(*t.next_code_token*) ∧ !isUserDefault(*t.next_code_token*) ∧ !invalidSecret(*t.next_code_token*)
CWE-319	Use of HTTP without TLS	(isVarAssign(*t*) ∨ isAtrAssign(*t*)) ∧ isHTTP(*t.next_code_token*) ∧ !inWhitelist(*t.next_code_token*)
CWE-546	Suspicious Comments	isComment(*t*) ∧ isSuspiciousWord(*t*)
CWE-326	Use of Weak Crypto. Algo.	(isVarAssign(*t.prev_code_token*) ∨ isAtrAssign(*t.prev_code_token*) ∨ isFunctionCall(*t.next_code_token*)) ∧ !isCheckSum(*t.prev_code_token*) ∧ isWeakCrypto(*t.next_code_token*)
CWE-284	Invalid IP Address Binding	(isVarAssign(*t*) ∨ isAtrAssign(*t*)) ∧ isInvalidIPBind(*t.next_code_token*)
CWE-258	Empty Password	(isVarAssign(*t*) ∨ isAtrAssign(*t*)) ∧ isPassword(*t.prev_code_token*) ∧ isEmptyPassword(t.prev_code_token)
CWE-250	Admin by default	(isVarAssign(*t*) ∨ isAtrAssign(*t*)) ∧ isNonSecret(*t.prev_code_token*) ∧ isUser(*t.prev_code_token*) ∧ !isPlaceholder(*t.next_code_token*) ∧ isAdmin(*t.next_code_token*)
CWE-1007	Homograph Attacks	(isVarAssign(*t*) ∨ isAtrAssign(*t*)) ∧ hasCyrillic(*t.next_code_token*)
CWE-521	Weak Password	(isVarAssign(*t*) ∨ isAtrAssign(*t*)) ∧ isPassword(*t.prev_code_token*) ∧ isStrongPwd(*t.next_code_token*)
CWE-829	Malicious Dependencies	isResource(*t*) ∧ isVersion(*t.prev_code_token*) ∧ isMalicious(*t.next_code_token*)

`inWhitelist(t.value)` verifies if the URL is in the list of configurable safe domains/whitelist. If the URL is in the whitelist, an aler
`isMalicious(t.value)` verifies if the software package version configured in the puppet script is in the database of malicious dep

cision of SLIC was 64%. By implementing the two design choices mentioned before, we increased precision by 12 per cent points—when comparing SLIC's precision in the *proportional* set (64%) with the precision of the first version of INFRASECURE in the same dataset (76%), Table **??**. As these changes were successful w.r.t. precision, we decided to implement the other improvements and conduct a new study with practitioners to collect more feedback about the tool and the anti-patterns covered.

4.6 Practitioners Evaluation

INFRASECURE was validated with practitioners experienced in security or configuration management technologies. We built an experiment to validate the warnings of the new tool. The experiment was shared with the Puppet communities on Slack (`puppetcommunity.slack.com`) and Reddit

(`r/puppet`). We found 14 participants by this mean. Later, we leveraged Prolific[25] [**?**] to gather more participants based on their experience and programming knowledge. In this experiment, a total of 339 warnings were validated by 131 practitioners. Furthermore, our improvements increased the precision of the tool from 28% to 83%. As illustrated in Figure **??**, we run two cycles of feedback collection and iteratively improve the tool with the feedback collected. This section describes 1) the methodology conducted with practitioners to validate the INFRASECURE warnings; and 2) the results obtained from running the practitioners' experiment.

4.6.1 Study Design

In this section, we detail how the validation study of INFRASECURE was designed, and the population leveraged to conduct it. INFRASECURE was improved based on the problems collected through the preliminary study and the validations with the maintainers of the software—which led to INFRASECURE v0.1.0. To validate the new tool, we surveyed practitioners with experience in security, configuration management tools and programming knowledge by following recent recommendations to run studies on Prolific [**?**]. After the pre-screening, the practitioners were asked to validate and give feedback on 3 different warnings generated by INFRASECURE.

Practitioners Recruitment. The participants were obtained using two distinct routes: 1) By sharing the study with online Puppet communities such as `puppetcommunity.slack.com` (slack) and `r/puppet` (reddit); 2) By using the Prolific platform to gather practitioners with experience in security, configuration management tools and programming skills. Both communities integrate a considerable amount of members: slack has over 9k members, and Reddit has around 4.7k members. However, only 14 members participated in our study. Therefore, we used Prolific to collect more practitioners with experience in security and configuration management tools outside of these two communities. Prolific participants were monetarily compensated for answering each survey, while the participants collected in the communities were not.

Pre-Screening. Prolific is a platform where you can find participants to perform online research. As recommended in research on recruiting practitioners for user studies on prolific [**?**], we performed a pre-screening of the population to collect adequate participants for this study, i.e., participants with security and configuration management experience; and programming knowledge. Prolific has filters dedicated to the industry where the participants work or belong. We sent the pre-screening survey to prolific users working in the following industries: "Computer and Electronics Manufacturing", "Information Services and Data Processing", "Product Development", "Research laboratories", "Scientific or Technical Services", "Software". The participants were asked to answer the following questions: 1) Do you have any kind of experience with configuration management tools? **Choices:** Puppet, Ansible, Terraform, Chef, Other; 2) Experience in Security (Number of Years); 3) Experience in Infrastructure as a Service (Number of Years); and three programming language questions about different puppet configurations. Due to space constraints, we do not present the questions here, but they are available in our replication package: `study/practitioners/pre-screening/puppet-study-form.pdf`.

[25] Prolific Platform: https://www.prolific.co/

We obtained a total of 431 responses from 8 different industries. Then, we ordered those participants by priority where priority is the count of experience in 1) at least one configuration management tool ($CMEXP$), 2) security ($SECEXP$), 3) infrastructure as a service ($INFRAEXP$); and 4) score in the programming questions ($SCORE$). Priority was calculated as follows $0.3 * ((CMEXP + SECEXP + INFRAEXP)/3) + 0.7 * (SCORE/3)$ and varies between 0 to 3. A priority of 3 means the participant is adequate for the study, whereas a priority of 0 means the participant is not adequate. For the validation study, we only invited participants with priority equal to or greater than 1.5—which represented 53% of the initial responses (227 out of 431 participants).

Validation Form. We built a form online to share with the Puppet communities and practitioners. The initial page of the form explains the study's goal and asks the participant for her profession/career, number of years of experience in security, and number of years of experience in infrastructure/puppet. The goal of the study is to validate the output of our new tool: INFRASECURE. Therefore, participants are required to validate 3 different warnings (one at each time). For each warning, the form presents a description of the issue and the piece of code where the issue is located (cf. Figure **??**). Participants have to evaluate the issue and provide their validation: "Yes, I agree", if the warning reports a security issue; "No, I disagree", if it reports a false security issue; or, "I'm not sure", when unsure. The participant can also provide additional feedback on the problem.

Warnings Dataset. For this experiment, we validated the output of 9 different rules (Table **??**), where the warnings for weak passwords and malicious dependencies were mostly validated in the second round of feedback collection. We ran the INFRASECURE over a total of 1050 GitHub projects—collected from the dataset used in the preliminary study (Section **??**). We created a uniform sample with 50 warnings per rule (i.e., a total of 450 warnings).

Metrics. We report the number of true positives (TPs), the number of false positives (TPs), the number of "Unsure" responses and Precision—calculated as described in Section **??**.

4.6.2 Results

We obtained a total of 131 participants: 74 in the first round of feedback; and 57 in the second round. Due to the lack of responses from participants, we were only able to validate 342 out of the 450 initial number of warnings. Table **??** shows the distribution of warnings and precision obtained for the final version of INFRASECURE (v1.1.0). At the end of this study, INFRASECURE reported a precision of 83%, where 54 of the warnings were False Positives.

Part of the feedback obtained in this experiment was documented in Section **??** and **??**. Table **??** shows the evolution of the tool's precision with the different iterations of feedback. In contrast to Table **??**, where we report the precision of v0.1.0 for the alerts validated by students; here, in Table **??**, we report the precision of v0.1.0 based on the practitioners' feedback, i.e., leveraging the alerts validated by practitioners (instead of students). It is important to note that the implementation of v0.1.0 focused on understanding variable and attribute assignments and reasoning about the token value to reduce the number of incorrect detections. These two improvements affected all checkers. The remaining versions of the tool focused on addressing specific false positives, extending the ruleset and

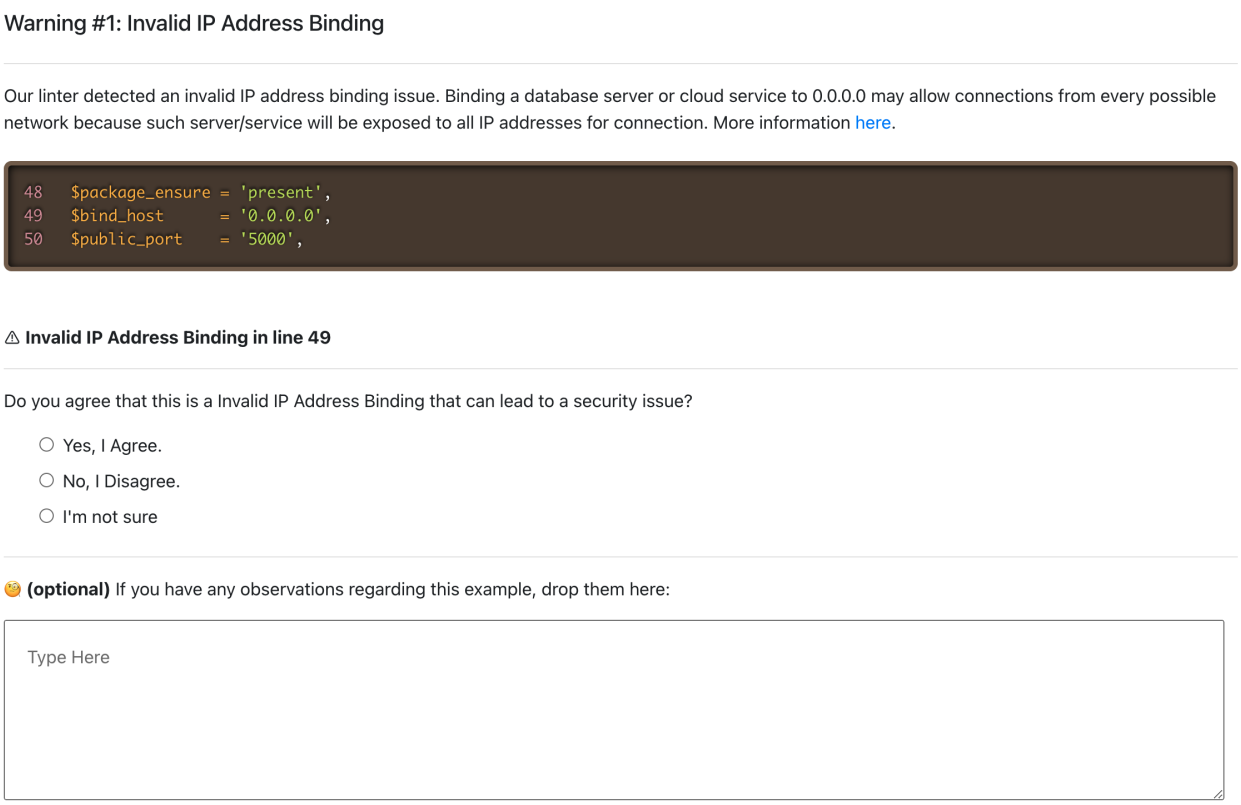


Figure 4.4: Example of the form presented to the practicioner for warning validation.

Table 4.8: Performance of INFRASECURE (v1.1.0). (Validation with Practitioners)

Rule	#TP	#FP	#Unsure	Precision
Hard-coded secrets	28	8	3	0.78
Use of HTTP without TLS	32	3	2	0.91
Suspicious Comments	16	15	7	0.52
Use of Weak Crypto. Algo.	33	3	6	0.92
Invalid IP Address Binding	26	8	6	0.77
Empty Password	33	3	1	0.92
Admin by default	30	6	6	0.83
Malicious Dependencies	25	6	3	0.81
Weak Password	32	2	0	0.94
Total	255	54	34	0.83

Table 4.9: Precision obtained in different cycles of feedback collection for INFRASECURE.

Participants	**version**	**Precision**
Research Team, Owners of OSS Projects, PuppetLabs, Voxpupuli	v0.1.0	76%
Practitioners (cycle 1)	v1.0.0	79%
Practitioners (cycle 2)	v1.1.0	83%

adding the safe domains feature. In the comparison provided in Table **??**, we observe an increase of precision—from 76% to 83%—by conducting different cycles of feedback collection. In addition, feedback was essential to extend the ruleset. This study with practitioners led us to create 3 new rules to detect weak passwords; typosquatting attacks; and malicious dependencies (being the last two the root causes of many supply chain attacks [**? ?**]).

Summary: Results show that working side-by-side with the community will help the authors of the tools develop better linters, as proposed before by a Google study [**?**]. Using this feedback approach, we improved the linter's precision and the final ruleset.

4.6.3 Discussion & Limitations

This paper reports our approach to improve the ruleset of an IaC security iteratively linter in different cycles of feedback collection. However, the tool can still be improved with more sophisticated techniques such as data-flow analysis, which would fulfil the following feedback: *In puppet, pre-defining a password as empty does not mean it is empty (e.g., `$ssl_password = ''`). Many times these variables are changed later. Thus, for each empty password,* INFRASECURE *verifies if the same variable was changed within the same file. If it was, then the linter will not raise an issue.*.

In addition, some engineers suggested that usernames should be only reported as hard-coded secrets when paired with a password/key. For this, we must match the different pairs of credentials in a puppet manifest. To sum up, there are still opportunities to improve the precision and recall of INFRASECURE. We reached out to owners of highly active GitHub projects that use Puppet reporting warnings detected by INFRASECURE. Two owners mentioned that since the apps are not in production, they did not consider the issues relevant. Even after improving the linter to detect the anti-patterns correctly, some problems are still not problematic. This happens because the linter does not have context regarding the software's usage, which will always be a source of False Positives. In the future, we will continue to search for solutions to make the linter more context-aware since this is a known problem of linters.

4.7 Ethical Standards and Compliance

This section discusses compliance with the ACM Policy for Research Involving Humans,[26] which ensures that the ethical and legal standards are met when research has human participants.

Informed Consent. One of the principles is to ensure that participants are informed about the fact that they are participating in a study. In our study, consent was collected differently for each experiment: for the first one, the research team agreed to participate in the study; for the OSS maintainers experiment, we used the puppet community slack to communicate and discuss the investigation with the maintainers; finally, for the practitioners' experiment, we asked survey participants if they agreed to participate in our different surveys at the beginning of the pre-screening phase.

Data Privacy. For all experiments, we ensured that the participants' private information was protected by not providing the participants' personal data (e.g., GitHub usernames of the OSS maintainers, prolific participants' names, ages, nationalities, etc.) in our replication package.

Spam. As mentioned in Section **??**, we carefully organized the issues to minimize the amount of messages sent to maintainers [**?** **?**]. Security smells of the same kind were all reported in a single GitHub issue. In addition, we designed the issues to be actionable by providing personalized fix suggestions and adding references that document the detected problems.

Full Disclosure in Security. Fully disclosing vulnerabilities on GitHub issues allows hackers to exploit unfixed vulnerabilities, creating risks for software users. Ideally, vulnerability disclosure should

[26]The ACM Policy for Research Involving Humans description is available at `https://www.acm.org/publications/policies/research-involving-human-participants-and-subjects` (Accessed 11 de outubro de 2023)

be performed confidentially. Yet, GitHub does not provide a feature to report them privately. Therefore, carefully performing full disclosure is accepted by OSS maintainers [**?** **?**]. Some OSS projects adopt security mailing lists. In those cases, disclosure should be performed through those mailing lists.

4.8 Threats to Validity

This section presents potential threats to the validity of this study.

Internal Validity: As with any implementation, the scripts that we developed to run the tools and collect the metrics reported in the paper are potentially not bug-free. However, the scripts and outputs are open-source for other researchers and potential users to check the validity of the results.

Construct Validity: A potential threat is the manual analysis of the warnings raised by SLIC in the Puppet scripts, which can potentially be mislabeled. We tried to mitigate this by running a kappa analysis between the two co-authors. For the experiments with the OSS maintainers, we inferred the validations of the alerts from their comments. Although both co-authors inferred the validations, and a kappa analysis was performed, we risk our inference being incorrect. It is also important to mention that even though we made an effort to collect feedback from experienced humans, their judgement can also not be 100% accurate, which can introduce error in the precision values reported.

External Validity: A potential threat to external validity is related to the fact that the set of Puppet scripts we have considered in this study may not accurately represent the whole set of vulnerabilities that can happen during development. We attempt to reduce the selection bias by gathering a large collection of real, openly available (hence, reproducible) Puppet scripts. Another potential threat is that we could have missed the latest updates to SLIC. To mitigate this risk, we contacted the authors of SLIC to confirm that the version available is true to the most recent one.

4.9 Related Work

As IaC has become popular and prevalent, researchers have dedicated efforts to improve its quality. Jiang et al. conducted an empirical study on Puppet scripts to gain a deep understanding of the characteristics of such scripts and how they evolve over time [**?**]. Bent et al. investigated the quality and maintainability aspects of Puppet scripts [**?**]. Furthermore, Rahman et al. proposed prediction models (based on text mining) to classify defective IaC scripts [**?**]. Palma et al. created a catalog of software metrics for IaC scripts [**?**]. In addition, recent work has been developed to detect malicious packages published on registry maintainers such as npm and ruby gems [**?**]. Building and introducing linters earlier in the software development life-cycle shift security left and decreases the probability of shipping malicious packages.

There are several linters available for security but only the subject of this paper, SLIC, focuses in IaC scripts for Puppet [**?**]. The authors started by demonstrating the linter in the context of Puppet scripts, and later, the authors reproduced the same study for Chef and Ansible and created new tools for those technologies [**?**]. A major issues with linters is their lack of precision [**? ? ? ? ? ?**]: low precision entails low reliability for developers. Previous research has shown the impact of this

issue on the developers' workflow and stressed it is essential to create precise tools; otherwise, the developers will not use them [**? ? ? ? ? ?**]. As mentioned before, this study aims to gain a better understanding of the current capabilities of the only IaC security linter for Puppet and shed some light on how to move forward.

4.10 Conclusions & Future Work

In this study, we observed through a comprehensive study that security linters for IaC scripts still need to be improved to be adopted by the industry. This paper leverages community expertise to address the challenge of improving the precision of such linting tools. We focused on precision as it is critically important in this domain—false security warnings can be very disruptive. More precisely, we interviewed professional developers of Puppet scripts to collect their feedback on the root causes of imprecision of the state-of-the-art security linter for Puppet. From that feedback, we developed a linter adjusting 7 rules of an existing linter ruleset and adding 3 new rules. We conducted a new study with 131 professional developers, showing an increase in precision from 28% to 83%. Following the findings of a Google study [**?**], we show that authors of linters can improve their own tools if they focus on the users' feedback. The takeaway messages of this paper are that (i) it is feasible to tune security linters to produce acceptable precision; and, that (ii) involving practitioners in discussions is an effective way to guide the improvement of those linters.

The observations that we made throughout this work pave the way for the following future work: extend INFRASECURE to detect other security vulnerabilities, integrate the tool with methods for automated patching, and port INFRASECURE to other configuration management tools.

4.11 Using Generative AI to improve the recall of a security linter

5

SAST Testing and Validation

Contents

Present the chapter content.

5.1 Section A

5.1.1 Subsection A

This would be a citation [**?**].

The COP defines the performance of the machine.

Heat Pump's performance is given by the COP_{HP}, a COP for heat pumps.

Now, an example on notation: Eu and $u_{v/s}$. Also C_D.

5.2 Section B

5.2.1 Subsection A

The model described can also be represented as

$$\dot{\mathbf{x}}(t) = \mathbf{Tz}(y),\ \mathbf{y}(0) = \mathbf{y}_0,\ z \geq 0 \tag{5.1}$$

where

$$\mathbf{A} = \begin{bmatrix} -(a_{12} + a_{10}) & a_{21} \\ a_{12} & -(a_{21} + a_{20}) \end{bmatrix},\ \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{5.2}$$

Also, using glossaries in the math environment, you can write

$$A = \frac{\dot{m}_v}{\rho u} \tag{5.3}$$

Note that A is not a.

5.2.2 Subsection B

Another example for the notation section: think about γ. And γ_p with a subscript.

Table 5.1: Dummy Table.

Vendor Name	Short Name	Commercial Name	Manufacturer
Text in Multiple Row	ABC	ABC®	ABC SA
	DEF	DEF®	DEF SA
	GHF	GHF®	GHF SA
Text in Single Row	IJK	IJK®	IJK SA
Frescos SA	LMN	LMN®	LMN SA
Carros Lda.	Text in Multiple Column		

6

Fixing Software Vulnerabilities Potentially Hinders Maintainability

Contents

Present the chapter content.

6.1 Introduction

Software quality is important because it is ultimately related to the overall cost of developing and maintaining software applications, security and safety [**?**]. Software quality characteristics include, but are not limited to functional correctness, reliability, usability, maintainability, evolvability and security. Security is an essential non-functional requirement during the development of software systems. In 2011, the International Organization for Standardization (ISO) issued an update for software product quality ISO/IEC 25010 considering *Security* as one of the main software product quality characteristics [**?**]. However, there is still a considerable amount of vulnerabilities being discovered and fixed, almost weekly, as disclosed by the Zero Day Initiative website[1].

Researchers found a correlation between the presence of vulnerabilities on software and code complexity [**?** **?**]. Security experts claim that complexity hides bugs that may result in security vulnerabilities [**?** **?**]. In practice, an attacker only needs to find one way into the system while a defender needs to find and mitigate all the security issues. Complex code is difficult to understand, maintain and test [**?**]. Thus, the task of a developer gets more challenging as the codebase grows in size and complexity. But the risk can be minimized by writing clean and maintainable code.

ISO describes *software maintainability* as "the degree of effectiveness and efficiency with which a software product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements" on software quality ISO/IEC 25010 [**?**]. Thereby, maintainable security may be defined, briefly, as the degree of effectiveness and efficiency with which software can be changed to mitigate a security vulnerability—corrective maintenance. However, many developers still lack knowledge on the best practices to deliver and maintain secure and high-quality software [**?** **?**]. In a world where zero-day vulnerabilities are constantly emerging, mitigation needs to be fast and efficient. Therefore, it is important to write maintainable code to support the production of more secure software—maintainable code is less complex and, consequently, less prone to vulnerabilities [**?** **?**]—and, prevent the introduction of new vulnerabilities.

As ISO does not provide any specific guidelines/formulas to calculate maintainability, we resort to Software Improvement Group (SIG[2])'s web-based source code analysis service Better Code Hub (BCH)[3] to compute the software compliance with a set of 10 guidelines/metrics to produce quality software based on ISO/IEC 25010 [**?**]. SIG has been helping business and technology leaders drive their organizational objectives by fundamentally improving the health and security of their software applications for more than 20 years. Their models are scientifically proven and certified [**?** **?** **?** **?**].

There are other well-known standards and models that have been proposed to increase software security: Common Criteria [**?**] which received negative criticism regarding the costs associated and poor technical evaluation; the OWASP Application Security Verification Standard (ASVS) [**?**]

[1] Zero Day Initiative website available at `https://www.zerodayinitiative.com/advisories/published/` (Accessed on 11 de outubro de 2023)

[2] SIG's website: `https://www.sig.eu/` (Accessed on 11 de outubro de 2023)

[3] BCH's website: `https://bettercodehub.com/` (Accessed on 11 de outubro de 2023)

which is focused only on web applications, and a model proposed by Xu et al. (2013) [**?**] for rating software security (arguably, it was one of the first steps taken by SIG to introduce security on their maintainability model). Nevertheless, our study uses BCH to provide an assessment of maintainability in software for the following reasons: BCH integrates a total of 10 different code metrics; and, code metrics were empirically validated in previous work [**? ? ? ?**].

Static analysis tools (SATs) have been built to detect software vulnerabilities automatically (e.g., FindBugs, Infer, and more). Developers use those tools to locate the issues in the code. However, while performing the patches to those issues, SATs cannot provide information on the quality of the patch. Improving software security is not a trivial task and requires implementing patches that might affect software maintainability. We hypothesize that some of these patches may have a negative impact on the software maintainability and, possibly, even be the cause of the introduction of new vulnerabilities—harming software reliability and introducing technical debt. Research found that 34% of the security patches performed introduce new problems and 52% are incomplete and do not fully secure systems [**?**]. Therefore, in this paper, we present an empirical study on the impact of patches of vulnerabilities on software maintenance across open-source software. We argue that tools that assess these type of code metrics may complement SATs with valuable information to help the developer understand the risk of its patch.

From a methodological perspective, we leveraged a dataset of 1300 security patches collected from open-source software. We calculate software maintainability before and after the patch to measure its impact. This empirical study presents evidence that changes applied in the codebases to patch vulnerabilities affect code maintainability. Results also suggest that developers should pay different levels of attention to different severity levels and classes of weaknesses when patching vulnerabilities. We also show that patches in programming languages such as, *C/C++*, *Ruby* and *PHP*, may have a more negative impact on software maintainability. Little information is known about the impact of security patches on software maintainability. Developers need to be aware of the impact of their changes on software maintainability while patching security vulnerabilities. The harm of maintainability can increase the time of response of future mitigations or even of other maintainability tasks. Thus, it is of utmost importance to find means to assist mitigation and reduce its risks. With this study, we intend to highlight the need for tools to assess the impact of patches on software maintainability [**?**]; the importance of integrating maintainable security in computer science curricula; and, the demand for better programming languages, designed by leveraging security principles [**? ?**].

This research performs the following main contributions:

- Evidence that supports the trade-off between security and maintainability: developers may be hindering software maintainability while patching vulnerabilities.
- An empirical study on the impact of security patches on software maintainability (per guideline, severity, weakness and programming language).
- A replication package with the scripts and data created to perform the empirical evaluation for reproducibility. Available online: `https://github.com/TQRG/maintainable-security`.

This paper is structured as follows: Section **??** introduces an example of a security patch of a known vulnerability found in the protocol implementation of OpenSSL[4]; Section **??** describes the methodology used to answer the research questions; Section **??** presents the results and discusses their implications; Section **??** elaborates on the implications developers should consider in the future; Section **??** enumerates the threats to the validity of this study; Section **??** describes the different work and existing literature in the field of study; and, finally, Section **??** concludes the main findings and elaborates on future work.

6.2 Motivation and Research Questions

As an example, consider the patch of the TLS state machine protocol implementation in OpenSSL**??** to address a memory leak flaw found in the way how OpenSSL handled TLS status request extension data during session renegotiation, and where a malicious client could cause a Denial-of-Service (DoS) attack via large Online Certificate Status Protocol (OCSP) Status Request extensions when OCSP stapling support was enabled. OCSP stapling, formally known as the TLS Certificate Status Request extension, is a standard for checking the revocation status of certificates.

This vulnerability is listed at the Common Vulnerabilities and Exposures dictionary as CVE-2016-6304 [5]. It is amongst the vulnerabilities studied in our research. The snippet, in Listing **??**, presents the changes performed on the *ssl/t1_lib.c* file[6] by the OpenSSL developers to patch the vulnerability. Every SSL/TLS connection begins with a handshake which is responsible for the negotiation between the two parties. The OSCP Status Request extension allows the client to verify the server certificate and enables a TLS server to include its response in the handshake. The problem in CVE-2016-6304 is a flaw in the logic of OpenSSL that does not handle memory efficiently when large OCSP Status Request extensions are sent each time a client requests renegotiation. This was possible because the OCSP responses IDs were not released between handshakes. Instead, they would be allocated again and again. Thus, if a malicious client does it several times it may lead to an unbounded memory growth on the server and, eventually, lead to a DoS attack through memory exhaustion.

The code changes performed to patch the CVE-2016-6304 vulnerability are presented in Listing **??**. The `sk_OCSP_RESPID_pop_free` function (**??**) removes any memory allocated to the OCSP response IDs (`OCSP_RESPID`s) from a previous handshake to prevent unbounded memory growth—which was not being performed before. After releasing the unbounded memory, the logic condition in **??** was shifted to **??** which is responsible for handling the application when no OCSP response IDS are allocated. After the patch, in the new version of the software, the condition is checked before the package processing instead of after. Thereby, the system avoids the increase of unbounded memory (and a potential DoS attack).

Patching this vulnerability seems a rudimentary task. Yet, a considerable amount of changes

[4]OpenSSL is a toolkit that contains open-source implementations of the SSL and TLS cryptographic protocols. Repository available at `https://github.com/openssl/openssl` (Accessed on 11 de outubro de 2023)

[5]CVE-2016-6304 details available at `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-6304` (Accessed on 11 de outubro de 2023)

[6]CVE-2016-6304 fix available at `https://github.com/openssl/openssl/commit/e408c09bbf7c3057bda4b8d20bec1b3a7771c15b` (Accessed on 11 de outubro de 2023)

were performed in the codebase which yielded a negative impact on software maintainability. While patching, the developer introduced 6 new lines in a method already with a large number of lines of code and introduced more complexity to the code with 2 new branch points, which disrupt two of the guidelines proposed by the Software Improvement Group (SIG) for building maintainable software [**?**]: *Write Short Units of Code* and *Write Simple Units of Code*.

```
1  static int ssl_scan_clienthello_tlsext(SSL *s, PACKET *pkt, int *al){
2   // [snip]
3  +  sk_OCSP_RESPID_pop_free(s->tlsext_ocsp_ids, OCSP_RESPID_free); ❶
4  +  if (PACKET_remaining(&responder_id_list) > 0) {
5  +     s->tlsext_ocsp_ids = sk_OCSP_RESPID_new_null();
6  +     if (s->tlsext_ocsp_ids == NULL) { ❷
7  +        *al = SSL_AD_INTERNAL_ERROR;
8  +        return 0;
9  +     }
10 +  } else {
11 +     s->tlsext_ocsp_ids = NULL;
12 +  }
13
14    while (PACKET_remaining(&responder_id_list) > 0) {
15      OCSP_RESPID *id;
16      PACKET responder_id;
17      const unsigned char *id_data;
18      if (!PACKET_get_length_prefixed_2(&responder_id_list, &responder_id) || PACKET_remaining(&
            responder_id) == 0) {
19          return 0;
20      }
21
22 -  if (s->tlsext_ocsp_ids == NULL
23 -      && (s->tlsext_ocsp_ids =
24 -      sk_OCSP_RESPID_new_null()) == NULL) { ❸
25 -    *al = SSL_AD_INTERNAL_ERROR;
26 -    return 0;
27 -  }
28
29   // [snip]
30  }
```

Listing 6.1: Patch provided by OpenSSL developers to the CVE-2016-6304 vulnerability on file ssl/t1_lib.c

Software maintainability is designated as the degree to which an application is understood, repaired, or enhanced. In this paper, our concern is to study whether while improving software security, developers are also hindering software maintainability. This is important because software maintainability is approximately 75% of the cost related to a project. To answer the following three research questions, we use two datasets of security patches [**?** **?**] to measure the impact of security patches on the maintainability of open-source software.

RQ1: What is the impact of security patches on the maintainability of open-source software? Often, security flaws require patching code to make software more secure. However, **there is no evidence yet of how security patches impact the maintainability of open-source software**. We hypothesize that developers tend to introduce technical debt in their software when patching software vulnerabilities because they tend not to pay enough attention to the quality of those patches. To address it, we follow the same methodology as previous research [**?**] and compute the maintainability of 1300 patches using the *Better Code Hub* tool. We present the maintainability impact by guideline/metric, overall score, severity, and programming language.

RQ2: Which weaknesses are more likely to affect open-source software maintainability? There are security flaws that are more difficult to patch than others. For instance, implementing secure authentication is not as easy as patching a cross-site scripting vulnerability since the latter can be fixed without adding new lines of code/complexity to the code. A typical fix for the cross-site scripting

vulnerability is presented in Listing **??**. The developer added the function `htmlentities` to escape the data given by the variable `$_['file']`. We hypothesize that security patches for different weaknesses can have different impacts on software maintainability. **Understanding which weaknesses are more likely to increase maintainability issues is one step toward bringing awareness to security engineers of what weaknesses need more attention.** The taxonomy of security patterns used to answer this question is the one provided by the Common Weakness Enumeration (CWE). *Weakness*, according to the Common Weakness Enumeration (CWE) glossary, is a type of code-flaw that could contribute to the introduction of vulnerabilities within that product. In this study, maintainability is measured separately for each weakness.

```
1     <p class='hint'>
2      <?php
3 -    if(isset($_['file'])) echo $_['file']
4 +    if(isset($_['file'])) echo htmlentities($_['file'])
5      ?>
6     </p>
```

Listing 6.2: Fix provided by `nextcloud/server` developers to a Cross-Site Scripting vulnerability

RQ3: What is the impact of security patches versus regular changes on the maintainability of open-source software? Performing a regular change/refactoring, for instance, to improve the name of a variable or function is different than performing a security patch. Therefore, we also computed the maintainability of random regular commits using the *Better Code Hub* tool—baseline. We use them to understand **how maintainability evolves when security patches are performed versus when they are not**.

6.3 Methodology

In this section, we discuss the methodology used to measure the impact of security patches on the maintainability of open-source software. The methodology comprises the following steps, as illustrated in Figure **??**.

1. Combine the datasets from related work that classify the activities of developers addressing security-oriented patches [**? ?**]. The duplicated patches were tossed.

2. Extract relevant data (e.g., owner and name of the repository, sha key of the vulnerable version, sha key of the fixed version) from the combined dataset containing 1300 security patches collected from open-source software available on GitHub.

3. Two baselines of regular changes were collected: *random-baseline* (for each security commit, a random change was collected from the same project) and *size-baseline* (for each security commit, a random change with the same size was collected from the same project). Our goal is to evaluate the impact of regular changes on the maintainability of open-source software.

4. Use the Software Improvement Group (SIG)'s web-based source code analysis service *Better Code Hub* (BCH) to quantify maintainability for both security and regular commits. BCH evaluates the codebase available in the default branch of a GitHub project. We created a tool that

pulls the codebase of each commit of our dataset to a new branch; it sets the new branch as the default branch; and, runs the BCH analysis on the codebase; after the analysis is finished, the tool saves the BCH metrics results to a cache file.

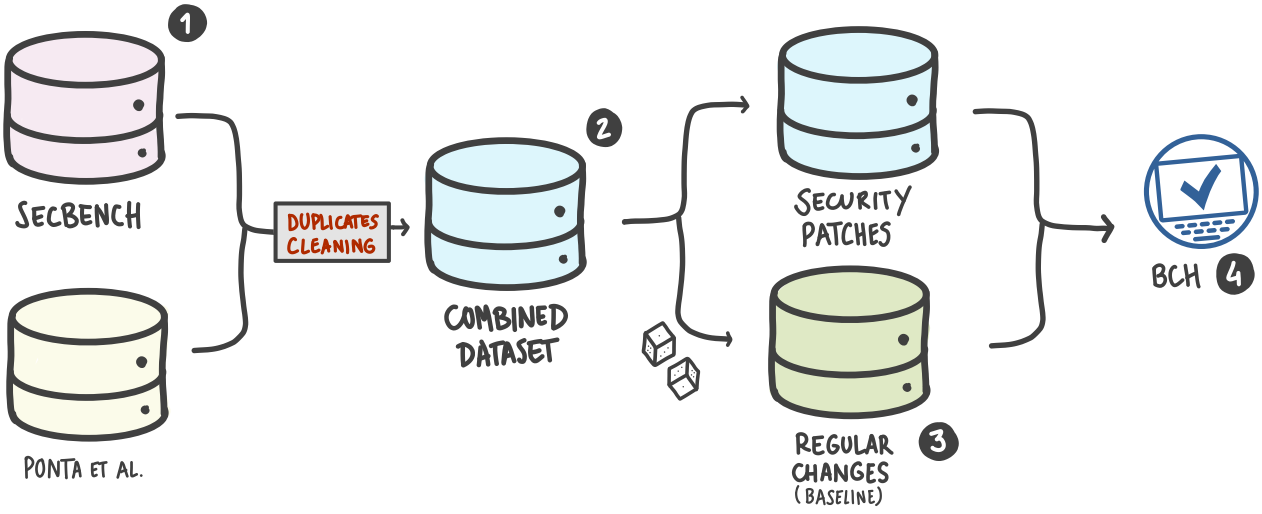


Figure 6.1: Study Methodology

6.3.1 Datasets

We use a combined dataset of 1300 security patches which is the outcome of mining and manually inspecting a total of 312 GitHub projects. The combined dataset integrates two different works: Secbench [**?** **?**] and Ponta et al. [**?**].

Reis and Abreu (2017**)** mined open-source software aiming at the extraction of real—created by developers—patches of security vulnerabilities to test and assess the performance of static analysis tools [**?** **?**] since using hand-seeded test cases or mutations can lead to misleading assessments of the capabilities of the tools [**?**]. The study yielded a dataset of 676 patches for 16 different security vulnerability types, dubbed as Secbench. The vulnerability types are based on the OWASP Top 10 of 2013 [**?**] and OWASP Top 10 of 2017 [**?**]. Each test case of the dataset is a triplet: the commit before the patching (*sha-p*), the commit responsible for the patching (*sha*), and the snippets of code that differ from one version to another (typically, called *diffs*)—where one can easily review the code used to fix the vulnerability.

Ponta et al. (2019**)** tracked the `pivotal.io` website for vulnerabilities from 2014 to 2019. For each new vulnerability, the authors manually searched for references to commits involved in the patch on the National Vulnerability Database (NVD) website. However, 70% of the vulnerabilities did not have any references to commits. Thus, the authors used their expertise to locate the commits in the repositories. This technique yielded a dataset of 624 patches [**?**] and 1282 commits—one patch

can have multiple commits assigned. To fit the dataset in our methodology, we located the first and last commits used to patch the vulnerability. For these cases, we used the GitHub API to retrieve the dates of the commits automatically. Then, for each patch, the group of commits was ordered from the oldest commit to the newest one. We assumed the last commit (newest one) as the fix (*sha*) and the parent of the first commit (oldest commit) as the vulnerable version (*sha-p*).

In this study, we focus on computing the maintainability of the commits before and after the security patching to evaluate if its impact was positive, negative, or none. The 1300 patches in the dataset were analyzed using the BCH toolset to calculate their maintainability reports. Due to the limitations of BCH (in particular, lack of language support and project size) and the presence of floss-refactorings, 331 patches were tossed—explained in more detail in Section **??**. The final dataset used in this paper comprises 969 security patches from 260 projects. We used the Common Weakness Enumeration (CWE) taxonomy to classify each vulnerability. For instance, the `Fix CVE-2014-1608: mc_issue_attachment_get SQL injection` (`00b4c17`[7]) is a *CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')*[8] according to the CWE taxonomy. We were able to classify a total of 866 patches using the CWE taxonomy: the CWE's for 536 patches were automatically scraped from the National Vulnerability Dataset (NVD); while the other 370 patches were manually classified by the authors following the *Research Concepts* CWE's list[9]. A total of 103 patches were not classified because we were not able to map the issue to any CWE with confidence due to the lack of quality information on the vulnerability/patch.

6.3.2 Security Patches vs. Regular Changes

Previous studies attempted to measure the impact of regular changes on open-source software maintainability [**?**]. However, there is no previous work focused on comparing the impact of security patches with regular changes on maintainability, only with bug-fixes [**?**]. We analyze the maintainability of regular changes—changes not related to security patches—and, use them as a baseline. The baseline dataset is generated from the security commits dataset, i.e., for each security commit in the dataset, we collect a random regular change from the same project. We created two different baselines: *random-baseline*, considering random changes and all their characteristics; and, *size-baseline*, considering also random changes but with an approximate size as security patches—we argue that comparing changes with considerably different sizes may be unfair.

6.3.2.A Random-Baseline

As for the security patches, for each regular change, we need the commit performing the regular change (*sha-reg*) and version of the software before the change (*sha-reg-p*). A random commit from the same project is selected for each security patch, *sha-reg*. The parent commit of *sha-reg* is the *sha-reg-p*.

[7] CVE-2014-1608 details available at `https://github.com/mantisbt/mantisbt/commit/00b4c17088fa56594d85fe46b6c6057bb3421102` (Accessed on 11 de outubro de 2023)

[8] CWE-89 details available at `https://cwe.mitre.org/data/definitions/89.html` (Accessed on 11 de outubro de 2023)

[9] Research Concepts list available at `https://cwe.mitre.org/data/definitions/1000.html`

6.3.2.B Size-Baseline

For the size-baseline, we also need to obtain the regular change (*sha-reg*) and the version of the software before the change (*sha-reg-p*). First, our tool calculates the *diff* between the security patch and its parent. Second, a random commit/regular change from the same project is selected, *sha-reg*. The *diff* between *sha-reg* and its parent (*sha-reg-p*) is calculated. Then, the regular change *diff* is compared to the security patch *diff*. Due to the complexity of some patches, it was not possible to find patches with the exact same number of added and deleted lines. Thus, we looked for an approximation.

The pair of the regular change (*sha-reg*) and its parent (*sha-reg-p*) is accepted if the *diff* size fits in the range size. This range widens every 10 attempts to search for a change with an approximate size. We originate the regular changes from the security commits to ensure that differences in maintainability are not a consequence of characteristics of different projects.

6.3.3 Bettter Code Hub

SIG—the company behind BCH—has been helping business and technology leaders drive their organizational objectives by fundamentally improving the health and security of their software applications for more than 20 years. The inner-workings of their SIG-MM model—the one behind BCH—are scientifically proven and certified [**? ? ? ?**].

BCH checks GitHub codebases against 10 maintainability guidelines [**?**] that were empirically validated in previous work [**? ? ? ?**]. SIG has devised these guidelines after many years of experience: analyzing more than 15 million lines of code every week, SIG maintains the industry's largest benchmark, containing more than 10 billion lines of code across 200+ technologies; SIG is the only lab in the world certified by TÜViT to issue ISO 25010 certificates[10]. BCH's compliance criterion is derived from the requirements for 4-star level maintainability (cf. ISO 25010) [**? ? ? ?**]. SIG performs the threshold calibration yearly on a proprietary data set to satisfy the requirements of TUViT to be a certified measurement model.

As BCH, other tools also perform code analysis for similar metrics. Two examples are Kiuwan and SonarCloud. However, Kiuwan does not provide the full description of the metrics it measures; and, SonarCloud although it provides a way of rating software maintainability, the variables description of their formula are not available. Both analyze less maintainability guidelines than BCH and do not have their inner workings fully and publicly described.

6.3.4 Maintainability Analysis

In this research, we follow a very similar methodology to the one presented in previous work on the maintainability of energy-oriented fixes [**?**]. The inner workings of BCH were proposed originally in 2007 [**?**] and suffered refinements later [**? ? ?**]. As said before, the web-based source code analysis service *Better Code Hub* (BCH) is used to collect the maintainability reports of the patches of

[10] Information available here: `https://www.softwareimprovementgroup.com/methodologies/iso-iec-25010-2011-standard/`

each project. Table **??** presents the 10 guidelines proposed by BCH's authors for delivering software that is not difficult to maintain [**?**] and, maps each guideline to the metric calculated by BCH. These guidelines are calculated using the metrics presented in [**?**] and are also briefly explained in Table **??**. During each guideline evaluation, the tool determines the compliance towards one guideline by establishing limits for the percentage of code allowed to be in each of the 4 risk severity levels (*low risk*, *medium risk*, *high risk*, and *very high risk*). If the project does not violate those thresholds, then the BCH considers that the code is compliant with a guideline. These thresholds are determined by BCH using their own data/experience—using open-source and closed software systems. If a project is compliant with a guideline, it means that it is at least 65% better than the software used by BCH to calculate the thresholds[11].

Table 6.1: Guidelines to produce maintainable code.

10 Guidelines	Description	Metric
Write Short Units of Code	Limit code units to 15 LOCs because smaller units are easier to understand, reuse and test them	**Unit Size:** % of LOCs within each unit [**?**]
Write Simple Units of Code	Limit branch points to 4 per unit because it makes units easier to test and modify	**McCabe Complexity:** # of decision points [**? ?**]
Write Code Once	Do not copy code because bugs tend to replicate at multiple places (inefficient and error-prone)	**Duplication:** % of redundant LOCs [**?**]
Keep Unit Interfaces Small	Limit the number of parameters to at most 4 because it makes units easier to understand and reuse	**Unit Interfacing:** # of parameters defined in a signature of a unit [**?**]
Separate Concerns in Modules	Avoid large modules because changes in loosely coupled databases are easier to oversee and execute	**Module Coupling:** # of incoming dependencies [**?**]
Couple Architecture Components Loosely	Minimize the amount of code within modules that are exposed to modules in other components	**Component Independence:** % of code in modules classified as hidden [**?**]
Keep Architecture Components Balanced	Balancing the number of components ease locating code and allow for isolated maintenance	**Component Balance:** Gini coefficient to measure the inequality of distribution between components [**?**]
Keep your code base Small	Reduce and avoid the system size because small products are easier to manage and maintain	**Volume:** # of LOCs converted to man-month/man-year [**?**]
Automate Tests	Test your code base because it makes development predictable and less risky	**Testability:** Ratings aggregation $-$ unit complexity, component independence and volume [**?**]
Write Clean Code	Avoid producing software with code smells because it is more likely to be maintainable in the future	**Code Smells:** # of Occurrences [**?**] (e.g., magic constants and long identifier names)

Figure **??** shows an example of the report provided by BCH for a project after finishing its evaluation. The example refers to the OpenSSL CVE-2016-6304 vulnerability patch— as described by Section **??**. This version of OpenSSL only complies with 1 out of 10 guidelines: *Write Clean Code*.

SIG defines *Units* as the smallest groups of code that can be maintained and executed independently [**?**] (e.g., methods and constructors in Java). One of the guidelines with which the project does not comply is the one presented in the report (cf. Figure **??**): *Write Simple Units of Code*. BCH analyzes this guideline based on the McCabe Complexity [**?**] to calculate the number of branch points

[11] Check the answer to *How can I adjust the threshold for passing/not passing a guideline?* at `https://bettercodehub.com/docs/faq` (Accessed on 11 de outubro de 2023)

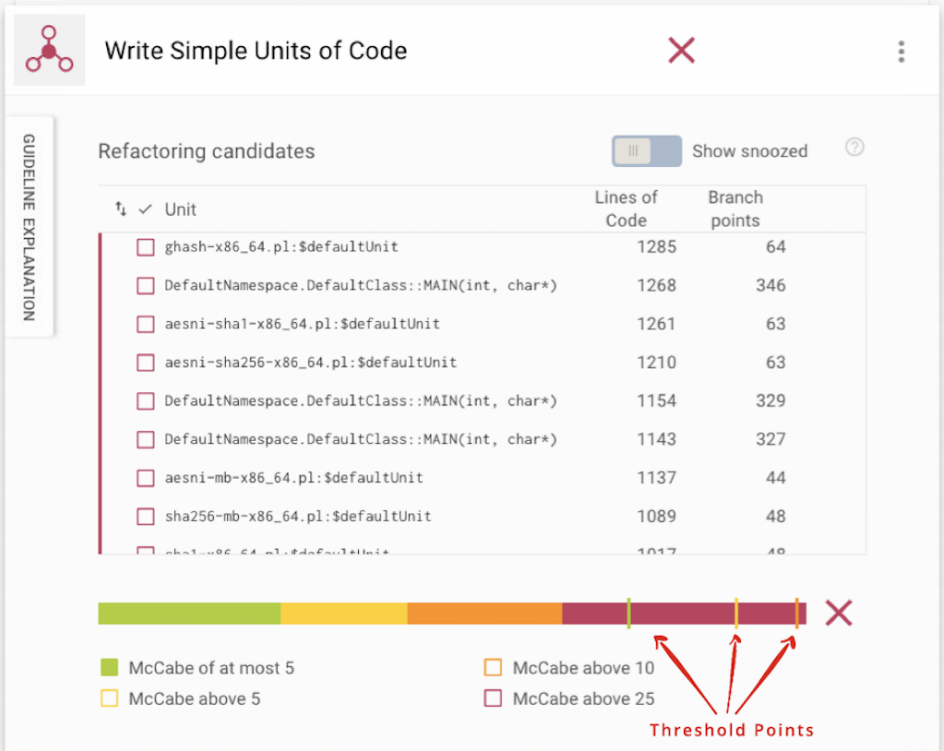


Figure 6.2: Maintainability report of OpenSSL's CVE-2016-6304 vulnerability patch for the guideline *Write Simple Units of Code* provided by *Better Code Hub*. This version of OpenSSL does not comply with the guideline in the example since the bars do not reach the threshold points. This example only complies with $^1/_{10}$ guidelines (*Write Clean Code*).

of a method. The bar at the bottom of the figure represents the top 30 units that violate the guideline, sorted by severity. The different severities of violating the guideline are indicated using colors, and there is a legend to help interpret them. The green bar represents the number of compliant branch points per unit (*at most* 5), i.e., the number of units are compliant with ISO 25010 [**?**]. Yellow, orange, and red bars represent units that do not comply with medium (*above* 5), high (*above* 10) and very high (*above* 25) severity levels. In the bar, there are marks that pinpoint the compliance thresholds for each severity level. If the green mark is somewhere in the green bar, it is compliant with a low severity level.

Aiming to analyze the impact of security patches, we use BCH to compute the maintainability of two different versions of the project (cf. Figure **??**):

- v_{s-1}, the version containing the security flaw, i.e., before the patch (*sha-p*);
- v_s, the version free of the security flaw, i.e., after the patch (*sha*);

Security patches can be performed through one commit (single-commit); several consecutive commits (multi-commits); or, commit(s) interleaved with more programming activities (floss-refactoring). Only 10.7% of the data points of our dataset involve more than one commit, the other 89.3% of the cases are single-commit patches. To mitigate the impact of floss-refactorings, we extracted and manually inspected a random sample with 25% of security patches from each dataset. From this sample, we identified 23 floss-refactorings. Most floss-refactoring patches include many changes making it

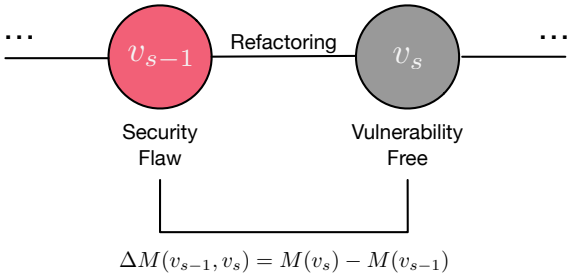


Figure 6.3: Maintainability difference for security commits.

difficult to understand which parts involve the security patch. Although we suspect that more floss-refactorings may occur, we argue that they occur in a small portion of the data.

Due to BCH limitations, in particular, lack of language support and project size by BCH, 308 data points were not analyzed and automatically disregarded from our study. After performing the BCH analysis and the maintainability calculations, we found the following limitations regarding two of BCH's guidelines:

1) For projects with large codebases, the results calculated for the *Keep Your Codebase Small* guideline were way above the limit set by BCH (20 person-years). We suspect this threshold may not be well-calibrated, and hence biasing our results. Thus, we decided not to consider this guideline in our research.

2) The *Automated Tests* guideline was also not considered since the tool does not include two of the most important techniques to security testing: vulnerability scanning and penetration testing. Instead, it only integrates unit testing.

The BCH tool does not compute the final score that our study needs to compare maintainability amongst different project versions. We follow previous work on measuring the impact of energy-oriented patches [**?**]. Cruz et al. (2019) proposed an equation to capture the distance between the current state of the project and the standard thresholds calculated by the BCH based on the insights provided in [**?**]. The equation provided in [**?**] considers that the size of project changes do not affect the maintainability, and that the distance to lower severity levels is less penalized than to the thresholds in high severity levels.

Given the violations for the BCH guidelines, the maintainability score is computed $M(v)$ as follows:

$$M(v) = \sum_{g \in G} M_g(v) \tag{6.1}$$

where G is the group of maintainability guidelines from BCH (Table **??**) and v is the version of the software under evaluation. $M(v) < 0$ indicates that version v is violating (some of) the guidelines, while $M(v) > 0$ indicates that version v is following the BCH guidelines. The maintenance for the guideline g, M_g for a given version of a project is computed as the summation of the compliance with the maintainability guideline for the given severity level (medium, high, and very high). The compliance for a severity level is calculated based on previous work, which calculates the number of lines of code that comply and not comply with the guideline at a given severity level [**?**]. In our

analysis, we compute the difference of maintainability between the security commit (v_s) and its parent commit (v_{s-1}), as illustrated in Figure **??**. Thus, we can determine which patches had a positive, negative, or null impact on the project maintainability.

6.3.5 Statistical Validation

To validate the maintainability differences in different groups of commits (e.g., baseline and security commits), we use the Paired Wilcoxon signed-rank test with the significance level $\alpha = 0.05$ [**?**]. In other words, we test the null hypothesis that the maintainability difference between pairs of versions v_{s-1}, v_s (i.e., before and after a security commit) come from the same distribution. Nevertheless, this test has a limitation: it does not consider the groups of commits with zero-difference maintainability. In 1959, Pratt improved the test to solve this issue, making the test more robust. Thus, we use a version of the Wilcoxon test that incorporates the cases where maintainability is equal to zero [**?**]. The Wilcoxon test requires a distribution size of at least 20 instances. To understand the effect-size, as advocated by the Common-language effect sizes, we compute the mean difference, the median of the difference, and the percentage of cases that reduce maintainability [**?**].

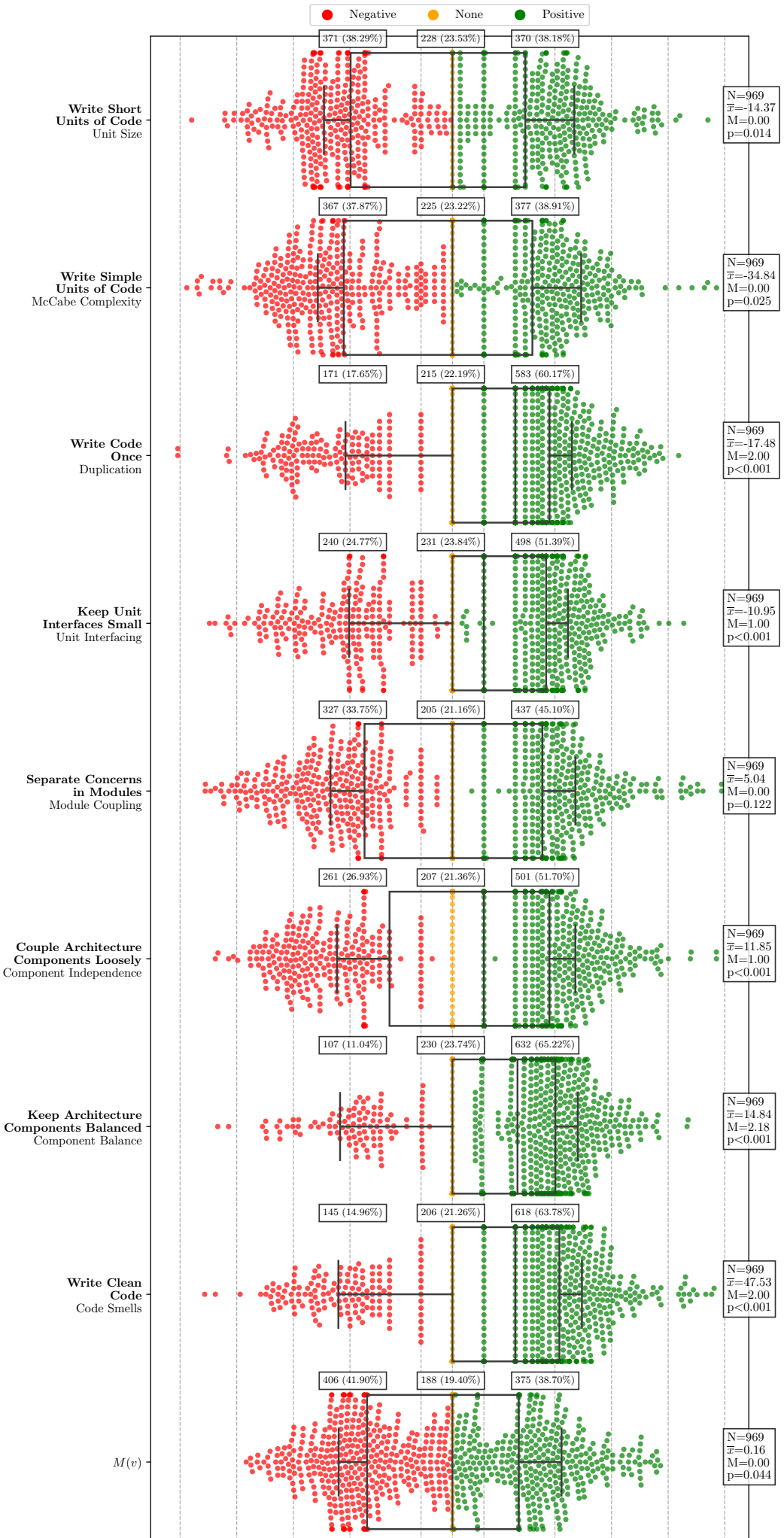
6.4 Results & Discussion

This study evaluates a total of 969 security patches and 969 regular changes from 260 distinct open-source projects. This section reports and discusses the results for each research question.

RQ1: What is the impact of security patches on the maintainability of open-source software? In *RQ1*, we report and discuss the impact of patches on open-source software maintainability under four groups: guideline, overall score, severity and programming language.

Guideline/Metric: Each patch performs a set of changes on the software's source code. These changes may have a different impact on the guidelines/metrics used to measure software maintainability. Figure **??** shows the impact of security patches on each guideline individually and the average impact on all guidelines together ($M(v)$). Under each guideline, it is stated the metric used for the calculations. For instance, for the *Write Short Units of Code* guideline, the metric used is *Unit Size*. Table **??** describes in more detail the metrics behind the guidelines. For each type of guideline, a swarm plot is presented to show the variability/dispersion of the results alongside the number of absolute and relative cases of each impact. Next to each type of guideline, it is presented the mean ($\bar{x}$) and median (M) of the maintainability difference and the p-value resulting from the Paired Wilcoxon signed-rank test. $M(v)$ is not a guideline but rather the average impact of all guidelines. Each point of the plot represents the impact of a security patch on software maintainability. Red means the impact was negative, i.e., the patch harmed maintainability. Yellow means the patch did not have any kind of impact on maintainability. Green means the impact was positive, i.e., the patch improved software maintainability.

Regarding the impact of security patches per guideline, we observe that 38.7% of the security patches have positive impact on software maintainability. However, we also see that patching vul-


Negative
None
Positive
Write Short Units of Code
Unit Size
371 (38.29%)
228 (23.53%)
370 (38.18%)
N=969
x̄=-14.37
M=0.00
p=0.014
Write Simple Units of Code
McCabe Complexity
367 (37.87%)
225 (23.22%)
377 (38.91%)
N=969
x̄=-34.84
M=0.00
p=0.025
Write Code Once
Duplication
171 (17.65%)
215 (22.19%)
583 (60.17%)
N=969
x̄=-17.48
M=2.00
p<0.001
Keep Unit Interfaces Small
Unit Interfacing
240 (24.77%)
231 (23.84%)
498 (51.39%)
N=969
x̄=-10.95
M=1.00
p<0.001
Separate Concerns in Modules
Module Coupling
327 (33.75%)
205 (21.16%)
437 (45.10%)
N=969
x̄=5.04
M=0.00
p=0.122
Couple Architecture Components Loosely
Component Independence
261 (26.93%)
207 (21.36%)
501 (51.70%)
N=969
x̄=11.85
M=1.00
p<0.001
Keep Architecture Components Balanced
Component Balance
107 (11.04%)
230 (23.74%)
632 (65.22%)
N=969
x̄=14.84
M=2.18
p<0.001
Write Clean Code
Code Smells
145 (14.96%)
206 (21.26%)
618 (63.78%)
N=969
x̄=47.53
M=2.00
p<0.001
M(v)
406 (41.90%)
188 (19.40%)
375 (38.70%)
N=969
x̄=0.16
M=0.00
p=0.044

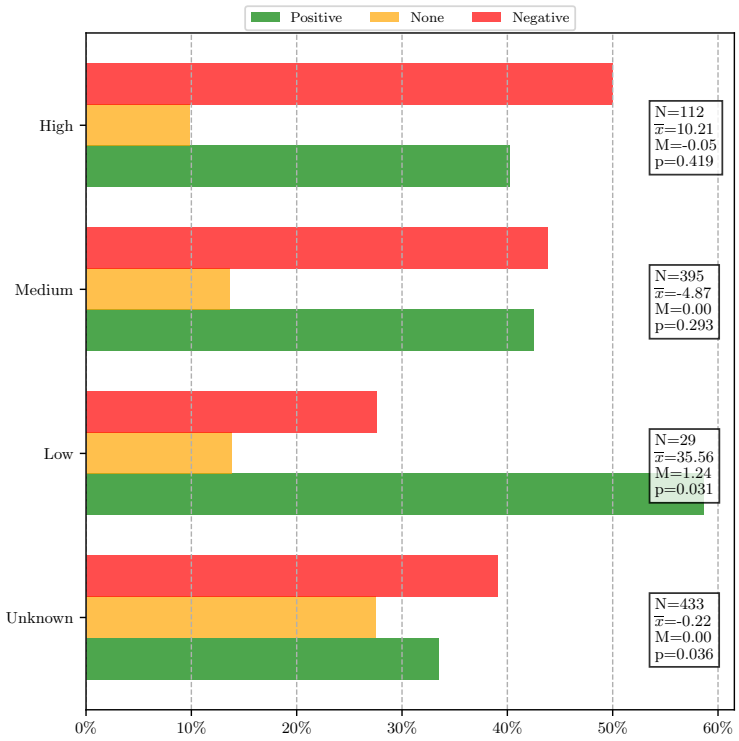


Figure 6.5: Maintainability difference by vulnerability severity.

nerabilities have a very significant number of negative cases per guideline—between 10% and 40%. *Write Short Units of Code* (38.3%), *Write Simple Units of Code* (37.9%), and *Separate Concerns in Modules* (33.8%) seem to be the most negatively affected guidelines. This may imply that developers, when patching vulnerabilities, have a hard time designing/implementing solutions that continue to respect the limit bounds of branch points and function/module sizes that are recommended by coding practices. Still, on respecting bound limits, developers also seem not to consider the limit of 4 parameters per function for the *Keep Unit Interfaces Small* guideline required by BCH, in 24.8% of the cases. This guideline is usually violated when the patch requires to input new information to a function/class, and developers struggle to use the *Introduce Parameter Object* patch pattern. Results do not provide statistical significance to the *Separate Concerns in Modules* guideline, i.e., results should be read carefully.

Software architecture is also affected while patching vulnerabilities. Both *Couple Architecture Component Loosely* and *Keep Architecture Components Balanced* guidelines suffer a negative impact of 26.9% and 11.0%, respectively. Component independence and balance are important to make it easier to find the source code that developers want to patch/improve and to understand how the high-level components interact with others. However, results may imply that developers forget to use techniques such as encapsulation to hide implementation details and make the system more modular.

The *Write Code Once* guideline results show that duplicated code increased in 17.7% ($171/969$) of the patches. Software systems typically have 9%-17% of cloned code [**?**]. Previous work showed a correlation between code smells and code duplication [**?**] which may also be reflected in the *Write Clean Code* guideline results. BCH reported new code smells for 15.0% ($145/969$) of the patches,

which according to previous work, may be the source of new software vulnerabilities [**? ?**] capable of harming the market value and economy of companies [**?**]. Developers should never reuse code by copying and pasting existing code fragments. Instead, they should create a method and call it every time needed. The *Extract Method* refactoring technique solves many duplication problems. This makes spotting and solving the issue faster because you only need to fix the method used instead of locating and fixing the issue multiple times. Clone detection tools can also help in locating the issues.

Overall Score ($M(v)$): Although overall patching vulnerabilities has a less negative impact on software maintainability guidelines, this is not reflected in the average impact of all guidelines ($M(v)$) as we can see in Figure **??**. Remember that each point of the plot represents the impact of a security patch on software maintainability. Red means the impact was negative, i.e., the patch harmed maintainability. Yellow means the patch did not have any kind of impact on maintainability. Green means the impact was positive, i.e., the patch improved software maintainability. The $M(v)$ plot shows that 406 (41.9%) cases have a negative impact on software maintainability. While 188 (19.4%) cases have no impact at all, and 375 (38.7%) have a positive impact on software maintainability. The larger number of negative cases may be explained by guidelines with higher concentrations of negative cases with higher amplitudes, such as *Write Short Units of Code*, *Write Simple Units of Code* and *Separate Concerns in Modules*—more red points on the left, being 0 the reference point. The resulting p-value of the Paired Wilcoxon signed-rank test for $M(v)$ is 0.044 (cf. Figure **??**). Since the p-value is below the significance level of 0.05, we argue that security patches may have a negative impact on the maintainability of open-source software.

Severity: Some of the vulnerabilities are identified with *Common Vulnerabilities and Exposure* (CVE) entries. We leveraged the *National Vulnerability Database* (NVD) website to collect their severity levels. In total, we retrieved severity scores for 536 vulnerabilities: 112 *High*, 395 *Medium* and 29 *Low*. Figure **??** presents the impact of security patches per severity level on the maintainability of open-source software. We observe that patches for *High* (50.0%) and *Medium* (43.8%) severity vulnerabilities hinder more the maintainability of software than *Low* (27.6%) severity vulnerabilities. Again, patches have a considerable negative impact on software maintainability—between 20% and 50%. Statistical significance was retrieved only for *Low* severity vulnerabilities, i.e., *Low* severity vulnerabilities may have more cases where software maintainability was improved than the other severity levels. However, results should not be disregarded because they somehow confirm the assumption that higher severity vulnerabilities patches may have a more negative impact on maintainability, i.e., high/medium severity vulnerabilities may need more attention than low severity while patching.

Programming Language: The impact on software maintainability per programming language was also analyzed (Figure **??**). We restrict this analysis to programming languages with at least 20 data points, as this is a requirement for the hypothesis tests. Thus, we compare the results for `C/C++`, `Ruby`, `Java`, `Objective C/C++`, `Python` and `PHP`, leaving `Groovy` out of the analysis. `C/C++`, `Ruby` and `PHP` are the programming languages with worse impact on maintainability, i.e., with the highest number of negative cases (46.5%, 46.5% and 38.6%, respectively). `Java` and `Python` seem to be less affected by patching, i.e., integrating a larger amount of cases with positive impact on

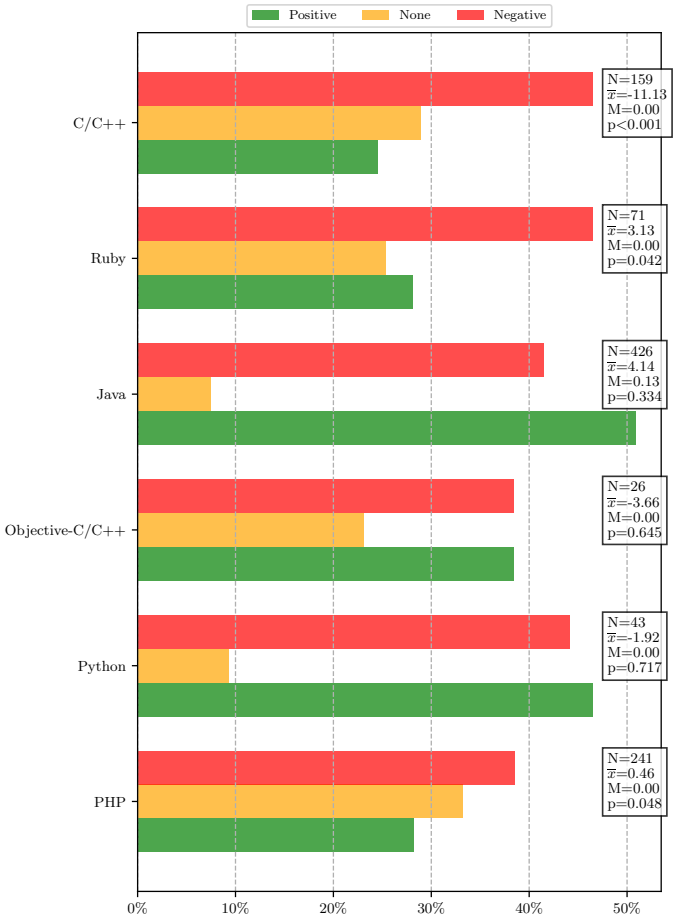


Figure 6.6: Maintainability difference by programming language.

maintainability (50.9% and 46.5%, respectively). But overall languages have a considerable amount of cases that negatively impact maintainability—between 35% to 50%—which confirms the need for better/more secure programming languages. Statistical significance was only retrieved for the `C/C++` ($p = 2.24\text{x}10^{-05}$), `Ruby` ($p = 0.041$) and `PHP` ($p = 0.048$) languages. Yet, data reports very interesting hints on the impact of programming languages on security patches.

We expected the negative impact for programming languages on maintainability to be more severe, as arguably, poor design of programming languages for security and the lack of best practices application by developers lead to more buggy/vulnerable code [**? ?**]. However, Figure **??** shows that only *C/C++* and *Ruby* have a significant negative impact approximate to 50% on maintainability. We suspect that these values are the result of project contributions policies (e.g., coding standards). In our dataset, $9/10$ projects with more contributors follow strict contribution policies for code standards.

Summary: Results show that developers may have a hard time following the guidelines and, consequently, hinder software maintainability while patching vulnerabilities; and that different levels of attention should be paid to each guideline. For instance, *Write Simple Units of Code* and *Write Short Units of Code* guidelines are the most affected ones. No statistical significance was observed for *Separate Concerns in Modules*. As shown in Figure **??**, there is statistical significance ($p = 0.044 < 0.05$) to support our findings: **security patches may have a negative impact on the maintainability of open-source software**. Therefore, tools such as BCH should be integrated into the CI/CD pipelines to help developers evaluate the risk of patches of hindering software maintainability—alongside Pull Requests/Code Reviews. Different severity vulnerabilities may need different levels of attention—high/medium vulnerabilities need more attention (cf. Figure **??**). However, statistical significance was only observed for low severity vulnerabilities. Better and more secure programming languages are needed. We observed statistical significance for *C/C++*, *Ruby* and *PHP* that support that security patches in those languages may hinder software maintainability (cf. Figure **??**).

RQ2: Which weaknesses are more likely to affect open-source software maintainability? In *RQ2*, we report/discuss the impact of security patches on software maintainability per weakness (CWE). We use the weakness definition and taxonomy proposed by the *Common Weakness Enumeration* (cf. Section **??**). Figure **??** shows three different charts. Figure **??**-*a*, presents the impact of the 969 patches grouped by the first level weaknesses from the *Research Concepts*[12] list. While the Figures **??**-*b* and **??**-*c* present the impact on maintainability for lower levels of weaknesses for the most prevalent weaknesses in Figure **??**-*a*: *Improper Neutralization* (CWE-707) and *Improper Control of a Resource Through its Lifetime* (CWE-664), respectively.

In Figure **??**-*a*, there is no clear evidence of the impact on maintainability per weakness. Yet, it is important to note that overall there is a very considerable number of cases that hinder maintainability—between 30% and 60%. The CWE-707 and CWE-664 weaknesses integrate the higher number of cases compared to the remaining ones: 295 (30.4%) data points and 318 (32.8%)

[12]Research Concepts is a tree-view provided by the Common Weakness Enumeration (CWE) website that intends to facilitate research into weaknesses. It is organized according to abstractions of behaviors instead of how they can be detected, their usual location in code, and when they are introduced in the development life cycle. The list is available here: `https://cwe.mitre.org/data/definitions/1000.html`

data points, respectively. Thus, we present an analysis of their sub-weaknesses on Figure **??**-*b* and Figure **??**-*c*, respectively.

Results shows that patching vulnerabilities may hinder the maintainability of open-source software in 4 different sub-weaknesses: *Improper Input Validation (CWE-20)*, *Information Exposure (CWE-200)*, *Missing Release of Memory after Effective Lifetime (CWE-401)* and *Path Traversal (CWE-22)*. Results also show that software maintainability is less negatively impacted when patching *Improper Restriction of XML External Entity Reference (CWE-611)*.

The impact of a patch depends on its complexity, i.e., if the patch adds complexity to the code base, it is probably affecting the software maintainability. *Cross-Site Scripting (CWE-79)* and *Improper Restriction of Operations within the Bounds of a Memory Buffer (CWE-119)* patches endure more cases with no impact on the open-source software maintainability. *SQL Injection (CWE-89)* patches equally hinder and improve software maintainability. These patches usually follow the same complexity as the CWE-79 patches. However, the three weaknesses have a considerable amount of cases that hinder the software maintainability—32.4%, 40.7% and, 41.1%, respectively—which should not be happening. Typically, CWE-79 vulnerabilities do not need extra lines to be fixed, as shown in Listing **??**—one simple `escape` function patches the issue. On the same type of fix, CWE-199 vulnerabilities may also be fixed without adding new source code (e.g., replacing the `strncpy` function with a more secure one `strlcpy` that checks if the buffer is null-terminated). However, some buffer overflows may be harder to fix and lead to more complex solutions (e.g., CVE-2016-0799[13]). As CWE-199 weaknesses, *Missing Release of Memory after Effective Lifetime (CWE-401)* can also be the cause of Denial-of-Service attacks and difficult to patch since it usually requires adding complexity to the program (cf. Section **??**).

Summary: Although results did not yield statistical significance, we show preliminary evidence that researchers and developers ought to pay more attention to maintainability when fixing the following types of weaknesses: *Improper Input Validation (CWE-20)*, *Information Exposure (CWE-200)*, *Missing Release of Memory after Effective Lifetime (CWE-401)* and *Path Traversal (CWE-22)*.

RQ3: What is the impact of security patches versus regular changes on the maintainability of open-source software? The impact of security and regular changes on software maintainability is presented in Figure **??**. In this section, we present a comparison of security patches with two different baselines of regular changes: *size-baseline*, a dataset of random regular changes with the same size as security patches—we argue that comparing changes with considerable different sizes may be unfair; and, *random-baseline*, a dataset of random regular changes.

Our hypothesis is that *security patches hinder more software maintainability than regular changes*. We have seen, previously, a deterioration in software maintainability when patching vulnerabilities: 41.9% (406) of patches suffered a negative impact, 38.7% (375) of patches remained the same, and 19.4% (188) of patches increased software maintainability. For regular changes, when considering the size of the changes (*size-basline*), we observe that the maintainability decreases in 27.0% (262) and increases in 30.5% (295) of the cases. But in contrast to security patches, the maintainability

[13]CVE-2016-0799 patch details available at `https://github.com/openssl/openssl/commit/9cb177301fdab492e4cfef376b28339afe3ef663` (Accessed on 11 de outubro de 2023)

of regular changes remains the same in 42.5% (412) of the cases, i.e., performing regular changes has a more positive impact than negative on maintainability. However, no statistical significance was obtained. Regular changes (*random-baseline*), with no size restrictions, are less prone to hinder software maintainability than security changes. About 34.4% (333) of the regular changes hinder software maintainability—less than in the security patches. For the *random-baseline*, statistical significance was retrieved ($p = 5.34\text{x}10^{-8}$).

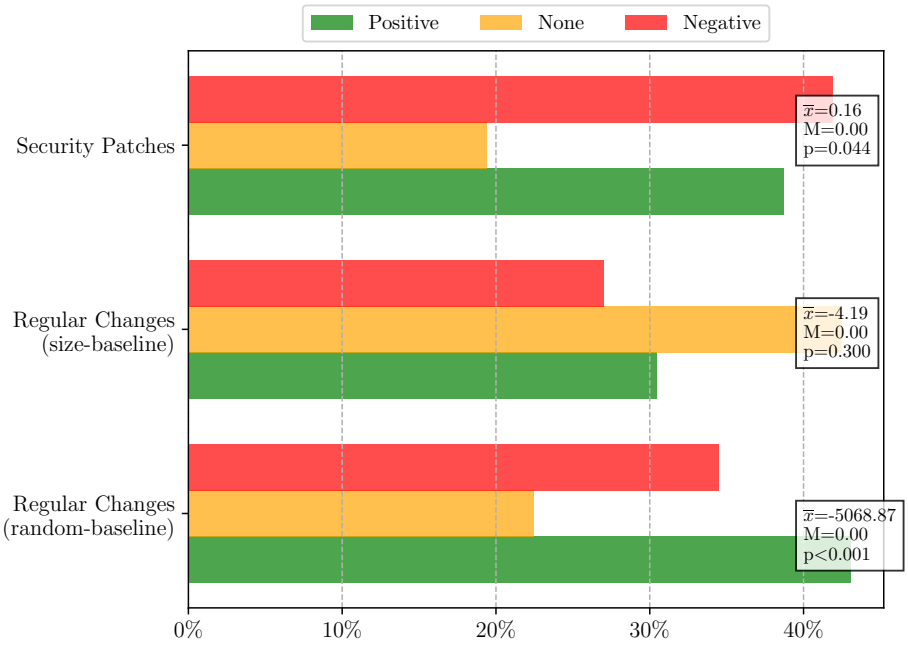


Figure 6.7: Maintainability difference of security patches versus regular changes.

Overall, the results for both baselines show that regular changes are less prone to hinder the software maintainability of open-source software. However, the *size-baseline* integrates a larger number of cases with no impact on software maintainability. We manually inspected a total of 25 cases from that distribution of regular changes with no impact on maintainability, and found that identifying regular changes with the same size as the security-related commit is limiting the type of regular commits being randomly chosen: input patches, variables or functions, type conversion (i.e., changes with no impact on the software metrics analyzed by BCH). We presume that this phenomenon lead to the significant number of cases where there is no impact on the software maintainability. On the other hand, identifying regular changes without any restrictions (*random-baseline*) shows that regular changes have a less negative impact on software maintainability when compared to security patches and that special attention should be given to security patches.

Summary: Security-related commits are observed to harm software maintainability, while regular changes are less prone to harm software maintainability. Thus, we urge the importance of adopting maintainability practices while applying security patches.

6.5 Study Implications

Our results show evidence that developers may have to reduce maintainability for the sake of security. We argue that developers should be able to patch and produce secure code without hindering the maintainability of their projects. But there are still concerns that need to be addressed and that this study brings awareness for:

Follow Best Practices: Developers are not paying attention to some quality aspects of their solutions/patches, as seen in Figure **??**, ending up harming software maintainability. We argue that developers should design and implement solutions that respect the limit bounds of branch points and function/module sizes that are recommended by best practices to avoid increasing the size, complexity, and dependencies of their patches. Developers should also keep function parameters below the recommended limit. It helps keep unit interfaces small and easy to use and understand. Patterns such as *Introduce Parameter Object* are useful to send information to a new function/class through an object and keep the number of parameters small and the information well-organized.

Security patches also harm the maintenance of software architecture. Maintaining the components independence and balance is important to make it easier to find the source code that developers want to patch/improve and to better understand how the high-level components interact with others. Applying *encapsulation* to hide implementation details and make the system more modular is a step forward not to hinder software architecture maintainability.

According to previous research, there is a correlation between code duplication and code smells [**?**]—duplicates are a source of regression bugs. BCH reports new code smells for 15% of the patches under study which supports previous research—34% of security patches introduce new problems [**?**]. Developers should never reuse code by copying and pasting existing code fragments. Instead, they should create a method and call it every time needed. The *Extract Method* refactoring technique solves many duplication problems. This makes spotting and solving the issue faster because you only need to fix one method instead of multiple vulnerabilities. Clone detection tools such as CPD can help on locating duplicates.

Prioritize High and Medium Severity: Previous research exhibits proof that developers prioritize higher impact vulnerabilities [**?**]. Our study shows that vulnerabilities of high and medium severity should be prioritized in software maintainability tasks.

Some Types of Vulnerabilities Need More Attention: Our study attempted to shed light on the impact of different types of vulnerabilities on software maintainability. Overall, all the CWEs under study present a negative impact over 30% on software maintainability. *Cross-Site Scripting* (CWE-79) and *Improper Restriction of Operations within the Bounds of a Memory Buffer* (CWE-119) are less prone to have an impact on open-source software maintainability. Developers should pay special attention to *Improper Input Validation* (CWE-20), *Information Exposure* (CWE-200), *Missing Release of Memory after Effective Lifetime* (CWE-401) and *Path Traversal* (CWE-22). However, more research should be performed to better understand the impact of each guideline on each CWE.

Tools for Patch Risk Assessment Wanted: Design debt of one guideline can lead to severe

impacts on the software quality [**?**]. Some software producers consider security as a first-class citizen while others do not. As mentioned in previous work, security is critical and should be considered as a default feature [**? ? ?**]. However, the lack of experts and awareness of developers for security while producing/patching software leads companies to ship low-quality software. Providing automated tools to developers to assess the risk of their patches is essential to help companies shipping software of higher quality. Bryan O'Sullivan, VP of Engineering at Facebook, advocated for new computer science risk models to detect vulnerabilities in scale and predict the level of security of the software under production in his talk "'Challenges in Making Software Work at Scale"' at FaceTAV'20.

Tools like Better Code Hub can complement static analysis (e.g., SonarQube, Codacy, ESLint, Infer, and more) to provide more informatio

6.6 Threats to Validity

This section presents the potential threats to the validity of this study.

Construct Validity: The formula to calculate the maintainability value ($M(v)$) was inferred based on the BCH's reports. The high amount of different projects and backgrounds may require other maintainability standards. However, BCH does use a representative benchmark of closed and open-source software projects to compute the thresholds for each maintainability guideline [**? ?**]. Maintainability is computed as the mean of all guidelines. Different software versions (vulnerable/fixed) of one vulnerability may have the same overall score and still be affected by different guidelines. Therefore, we provide an analysis per guideline, and our results are all available on GitHub for future reproductions and deeper analysis.

Internal Validity: The security patches dataset provided by previous work [**?**] was collected based on the messages of GitHub commits produced by project developers to classify the changes performed while patching vulnerabilities. This approach discards patches that were not explicit in commits messages. We assume that patches were performed using a single commit or several sequentially. The perspective that a developer may quickly perform a patch and later proceed to the refactor is not considered. We assume that all patches were only performed once. Depending on the impact of the vulnerability in the system, some vulnerabilities may have more urgency to be patched than others. For instance, a vulnerability performing a Denial-of-Service attack that usually brings entire systems down may be more urgent to patch than a cross-site scripting vulnerability which generally does not have an impact on the execution of the system but rather on the data accessibility. We manually inspected 25.1% of the security patches looking for floss-refactorings—122 from each dataset. We did find 23 cases we argue to be floss-refactorings and toss them to minimize the impact of this threat.

Baseline commits are retrieved randomly from the same project as the security patch. This approach softens the differences that may result from the characteristics of each project. However, maintainability may still be affected by the developers' experience, coding style, and software contribution policies which are not evaluated in this study. Furthermore, this evaluation considers that 969

regular commits—any kind of commit—are enough to alleviate random irregularities in the maintainability differences of the baseline.

External Validity: The BCH tool uses private and open-source data to determine the thresholds for each guideline. We only analyze patches of open-source software. Thus, our findings may not extend to private/non-open source software. Different programming languages may require different coding practices to address software safety. The dataset comprises more commits in Java, i.e., the dataset may not be representative of the population regarding programming languages. For both datasets, manual validation of the message of the commits was performed. Only commits in English were considered. Thus, our approach does not consider patches in any other language but English.

6.7 Related Work

Many studies have investigated the relationship between patches and software quality. Previous work focused on object-oriented metrics has evaluated the impact of patches and exhibited proof that quantifying the impact of patches on maintainability may help to choose the appropriate patch type [**?**]. In contrast to this work, Hegedus et al. [**?**] did not select particular metrics to assess the effect of patches. Instead, statistical tests were used to find the metrics that have the potential to change significantly after patches.

Researchers performed a large-scale empirical study to understand the characteristics of security patches and their differences against bug fixes [**?**]. The main findings were that security patches are smaller and less complex than bug fixes and are usually performed at the function level. Our study compares the impact of security patches on software maintainability with the impact of regular changes.

Studying the evolution of maintainability issues during the development of Android apps, Malavolta et al. (2018) [**?**] discovered that maintainability decreases over time. Palomba et al. (2018) [**?**] exhibits proof that code smells should be carefully monitored by programmers since there is a high correlation between maintainability aspects and proneness to changes/faults. In 2019, Cruz et al. [**?**] proposed a formula to calculate maintainability based on the BCH's guidelines and measured the impact of energy-oriented fixes on software maintainability. Recent work proposed a new maintainability model to measure fine-grained code changes by adapting/extending the BCH model [**?**]. Our work uses the same base model (SIG-MM) but considers a broader set of guidelines. Moreover, we solely focus on evaluating the impact of security patches on software maintainability.

Researchers investigated the relationship between design patterns and maintainability [**?**]. However, other studies show that the use of design patterns may introduce maintainability issues into software [**?**]. Yskout et. al did not detect if the usage of design patterns has a positive impact but concluded that developers prefer to work with the support of security patterns [**?**]. The present work studies how security weaknesses influence maintainability for open-source software.

There are studies that investigated the impact of programming languages on software quality [**?** **?**]. The first one shows that some programming languages are more buggy-prone than others.

However, the authors of the second one could not reproduce it and did not obtain any evidence about the language design impact. Berger et al. (2019) [**?**] tried to reproduce [**?** **?**] and identified flaws that throw into distrust the previously demonstrated a correlation between programming language and software defects. Our work studies how security patches affect software quality based on the code maintainability analysis and provides shows that programming languages may have an impact on maintainability.

6.8 Conclusion and Future Work

This work presents an empirical study on the impact of 969 security patches on the maintainability of 260 open-source projects. We leveraged Better Code Hub reports to calculate maintainability based on a model proposed in previous work [**?** **?**]. Results show evidence of a trade-off between security and maintainability, as 41.9% of security patches yielded a negative impact. Hence, developers may be hindering software maintainability while patching vulnerabilities. We also observe that some guidelines and programming languages are more likely to be affected than others. The implications of our study are that changes to codebases while patching vulnerabilities need to be performed with extra care; tools for patch risk assessment should be integrated into the CI/CD pipeline; computer science curricula need to be updated; and more secure programming languages are necessary.

As future work, the study can be extended in several directions: investigate which guidelines affect most the maintainability per weakness; check if vulnerability patches are followed by new commits and how much time does it take to do it; expand our methodology with other software quality properties; validate these findings with closed/private software; and, expand this analysis to other quality standards.

7

Conclusions and Future Work

Title of AppendixA

A.1 section 1

texto...