Fixing vulnerabilities potentially hinders maintainability

Reis, Sofia; Abreu, Rui; Cruz, Luis

DOI
10.1007/s10664-021-10019-z

Publication date
2021

Document Version
Final published version

Published in
Empirical Software Engineering

Citation (APA)
Reis, S., Abreu, R., & Cruz, L. (2021). Fixing vulnerabilities potentially hinders maintainability. *Empirical Software Engineering*, *26*(6), Article 127. https://doi.org/10.1007/s10664-021-10019-z

Empirical Software Engineering (2021) 26:127
https://doi.org/10.1007/s10664-021-10019-z

Fixing vulnerabilities potentially hinders maintainability

Sofia Reis[1] · Rui Abreu[2] · Luis Cruz[3]

Accepted: 19 May 2021Published online: 22 September 2021

Abstract
Security is a requirement of utmost importance to produce high-quality software. However, there is still a considerable amount of vulnerabilities being discovered and fixed almost weekly. We hypothesize that developers affect the maintainability of their codebases when patching vulnerabilities. This paper evaluates the impact of patches to improve security on the maintainability of open-source software. Maintainability is measured based on the Better Code Hub's model of 10 guidelines on a dataset, including 1300 security-related commits. Results show evidence of a trade-off between security and maintainability for 41.90% of the cases, i.e., developers may hinder software maintainability. Our analysis shows that 38.29% of patches increased software complexity and 37.87% of patches increased the percentage of LOCs per unit. 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 integrate into the CI/CD pipeline; computer science curricula needs to be updated; and, more secure programming languages are necessary.

Keywords Software security · Software maintenance · Open-source software

1 Introduction

Software quality is important because it is ultimately related to the overall cost of developing and maintaining software applications, security and safety (Slaughter et al. 1998).

Communicated by: Burak Turhan

✉ Sofia Reis
sofia.o.reis@tecnico.ulisboa.pt

Rui Abreu
rui@computer.org

Luis Cruz
L.Cruz@tudelft.nl

[1] INESC-ID and IST, University of Lisbon, Lisbon, Portugal

[2] INESC-ID and FEUP, University of Porto, Porto, Portugal

[3] Delft University of Technology, Delft, The Netherlands

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 (International Organization for Standardization 2011). 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 (Shin et al. 2010; Chowdhury and Zulkernine 2010). Security experts claim that complexity hides bugs that may result in security vulnerabilities (McGraw 2004; Schneier 2006). 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 (McCabe 1976). 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 (International Organization for Standardization 2011). 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 (Pothamsetty 2005; Acar et al. 2017). 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 (Shin et al. 2010; Chowdhury and Zulkernine 2010)— 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 (Visser 2016). 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 (Heitlager et al. 2007; Alves et al. 2010; Alves et al. 2011; Baggen et al. 2012).

There are other well-known standards and models that have been proposed to increase software security: Common Criteria (2009) which received negative criticism regarding the costs associated and poor technical evaluation; the OWASP Application Security Verification Standard (ASVS) (The OWASP Foundation 2009) 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

[1]Zero Day Initiative website available at https://www.zerodayinitiative.com/advisories/published/ (Accessed on September 20, 2021)

[2]SIG's website: https://www.sig.eu/ (Accessed on September 20, 2021)

[3]BCH's website: https://bettercodehub.com/ (Accessed on September 20, 2021)

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 (Bijlsma et al. 2012; Malavolta et al. 2018; Cruz et al. 2019; di Biase et al. 2019).

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 (Li and Paxson 2017). 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 (Maruyama and Tokoda 2008); the importance of integrating maintainable security in computer science curricula; and, the demand for better programming languages, designed by leveraging security principles (Kurilova et al. 2014; Nistor et al. 2013).

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 2 introduces an example of a security patch of a known vulnerability found in the protocol implementation of OpenSSL[4]; Section 3 describes the methodology used to answer the research questions; Section 4 presents the

[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 September 20, 2021)

results and discusses their implications; Section 5 elaborates on the implications developers should consider in the future; Section 6 enumerates the threats to the validity of this study; Section 7 describes the different work and existing literature in the field of study; and, finally, Section 8 concludes the main findings and elaborates on future work.

2 Motivation and Research Questions

As an example, consider the patch of the TLS state machine protocol implementation in OpenSSL[4] 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 1, 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 1. The `sk_OCSP_RESPID_pop_free` function (❶) removes any memory allocated to the OCSP response IDs (`OCSP_RESPIDs`) from a previous handshake to prevent unbounded memory growth—which was not being performed before. After releasing the unbounded memory, the logic condition in ❸ was shifted to ❷ 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 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 (Visser 2016): *Write Short Units of Code* and *Write Simple Units of Code*.

[5]CVE-2016-6304 details available at http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-6304 (Accessed on September 20, 2021)

[6]CVE-2016-6304 fix available at https://github.com/openssl/openssl/commit/e408c09bbf7c3057bda4b8d20bec1b3a7771c15b (Accessed on September 20, 2021)

```
static int ssl_scan_clienthello_tlsext(SSL *s, PACKET *pkt,
    int *al){
 // [snip]
+  sk_OCSP_RESPID_pop_free(s->tlsext_ocsp_ids,
    OCSP_RESPID_free); ❶
+  if (PACKET_remaining(&responder_id_list) > 0) {
+     s->tlsext_ocsp_ids = sk_OCSP_RESPID_new_null();
+     if (s->tlsext_ocsp_ids == NULL) { ❷
+        *al = SSL_AD_INTERNAL_ERROR;
+        return 0;
+     }
+  } else {
+     s->tlsext_ocsp_ids = NULL;
+  }

   while (PACKET_remaining(&responder_id_list) > 0) {
     OCSP_RESPID *id;
     PACKET responder_id;
     const unsigned char *id_data;
     if (!PACKET_get_length_prefixed_2(&responder_id_list, &
         responder_id) || PACKET_remaining(&responder_id) ==
         0) {
           return 0;
     }

-  if (s->tlsext_ocsp_ids == NULL
-      && (s->tlsext_ocsp_ids =
-      sk_OCSP_RESPID_new_null()) == NULL) { ❸
-    *al = SSL_AD_INTERNAL_ERROR;
-    return 0;
-  }

 // [snip]
 }
```

Listing 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 (Reis and Abreu 2017a; Ponta et al. 2019) 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 (Cruz et al. 2019) 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,

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

Listing 2 Fix provided by `nextcloud/server` developers to a Cross-Site Scripting vulnerability

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 2. 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.

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**.

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 Fig. 1.

1. Combine the datasets from related work that classify the activities of developers addressing security-oriented patches (Reis and Abreu 2017b; Ponta et al. 2019). 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;

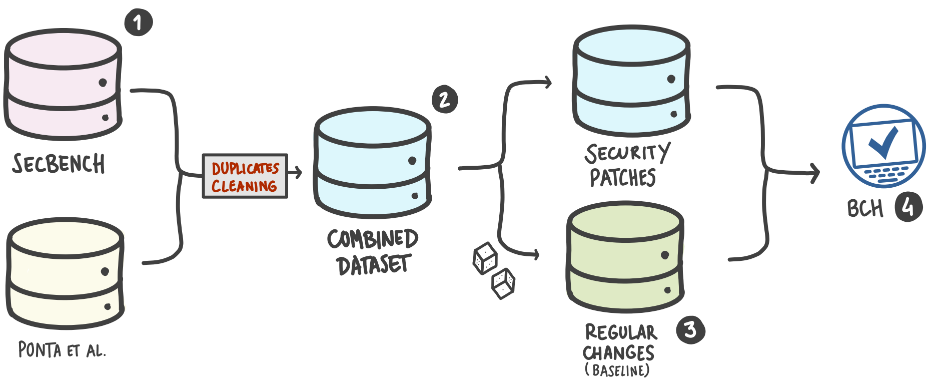


Fig. 1 Study Methodology

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.

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 Reis and Abreu (2017a, b), and Ponta et al. (2019).

Reis and Abreu (2017a) 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 Reis and Abreu (2017a, b) since using hand-seeded test cases or mutations can lead to misleading assessments of the capabilities of the tools (Just et al. 2014). 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 (Foundation 2017a) and OWASP Top 10 of 2017 (Foundation 2017b). 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 (Ponta et al. 2019) 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 3.4. 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.

3.2 Security Patches vs. Regular Changes

Previous studies attempted to measure the impact of regular changes on open-source software maintainability (Hegedűs et al. 2018). However, there is no previous work focused on comparing the impact of security patches with regular changes on maintainability, only with bug-fixes (Li and Paxson 2017). 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.

3.2.1 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*.

3.2.2 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

[7]CVE-2014-1608 details available at https://github.com/mantisbt/mantisbt/commit/00b4c17088fa56594d85fe46b6c6057bb3421102 (Accessed on September 20, 2021)

[8]CWE-89 details available at https://cwe.mitre.org/data/definitions/89.html (Accessed on September 20, 2021)

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

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.

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 (Heitlager et al. 2007; Alves et al. 2010; Alves et al. 2011; Baggen et al. 2012).

BCH checks GitHub codebases against 10 maintainability guidelines (Visser 2016) that were empirically validated in previous work (Bijlsma et al. 2012; Malavolta et al. 2018; Cruz et al. 2019; di Biase et al. 2019). 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) (Alves et al. 2010; Alves et al. 2011; Baggen et al. 2012; Visser 2016). 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.

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 (Cruz et al. 2019). The inner workings of BCH were proposed originally in 2007 (Heitlager et al. 2007) and suffered refinements later (Alves et al. 2010; Alves et al. 2011; Baggen et al. 2012). 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 each project. Table 1 presents the 10 guidelines proposed by BCH's authors for delivering software that is not difficult to maintain (Visser 2016) and, maps each guideline to the metric calculated by BCH. These guidelines are calculated using the metrics presented in Visser (2020) and are also briefly explained in Table 1. 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

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

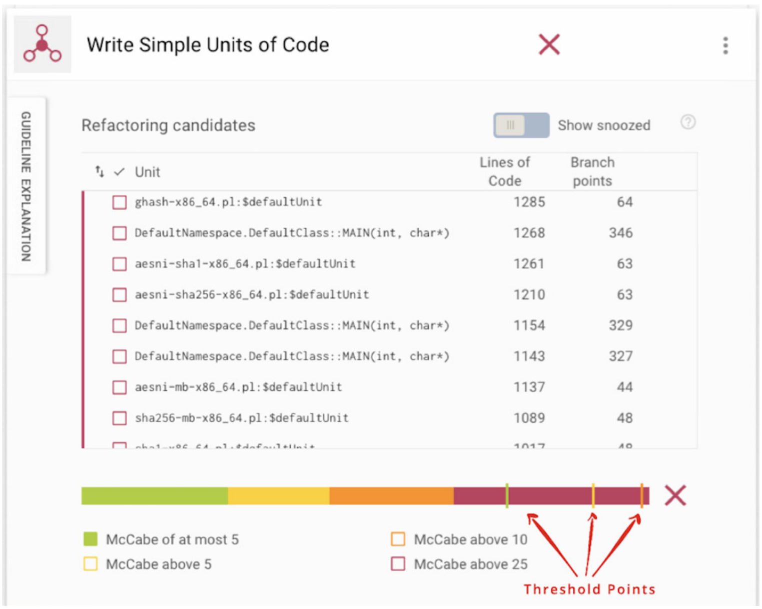
Table 1 Guidelines to produce maintainable code

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 (Visser 2020)
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 (McCabe 1976; Visser 2020)
Write Code Once	Do not copy code because bugs tend to replicate at multiple places (inefficient and error-prone)	**Duplication:** % of redundant LOCs (Visser 2020)
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 (Visser 2020)
Separate Concerns in Modules	Avoid large modules because changes in loosely coupled databases are easier to oversee and execute	**Module Coupling:** # of incoming dependencies (Visser 2020)
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 (Visser 2020)
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 (Visser 2020)
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 (Visser 2020)
Automate Tests	Test your code base because it makes development predictable and less risky	**Testability:** Ratings aggregation – unit complexity, component independence and volume (Visser 2016)
Write Clean Code	Avoid producing software with code smells because it is more likely to be maintainable in the future	**Code Smells:** # of Occurrences (Visser 2016) (e.g., magic constants and long identifier names)

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]

Figure 2 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

[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 September 20, 2021)



Unit	Lines of Code	Branch points
ghash-x86_64.pl:$defaultUnit	1285	64
DefaultNamespace.DefaultClass::MAIN(int, char*)	1268	346
aesni-sha1-x86_64.pl:$defaultUnit	1261	63
aesni-sha256-x86_64.pl:$defaultUnit	1210	63
DefaultNamespace.DefaultClass::MAIN(int, char*)	1154	329
DefaultNamespace.DefaultClass::MAIN(int, char*)	1143	327
aesni-mb-x86_64.pl:$defaultUnit	1137	44
sha256-mb-x86_64.pl:$defaultUnit	1089	48
sha1-x86_64.pl:$defaultUnit	1017	48

Fig. 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 $\frac{1}{10}$ guidelines (*Write Clean Code*)

described by Section 2. 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 (Visser 2016) (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. Fig. 2): *Write Simple Units of Code*. BCH analyzes this guideline based on the McCabe Complexity (McCabe 1976) to calculate the number of branch points 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 (International Organization for Standardization 2011). 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. Fig. 3):

– 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

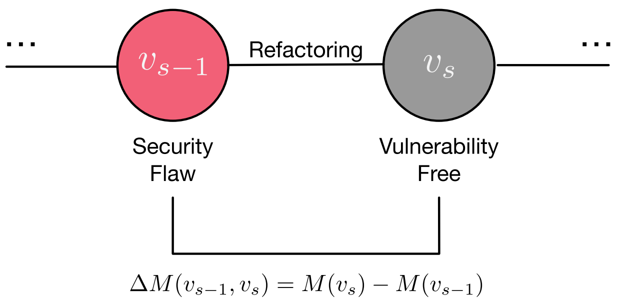


Fig. 3 Maintainability difference for security commits

(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 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). 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 Olivari (2018). The equation provided in Cruz et al. (2019) 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:

$$M(v) = \sum_{g \in G} M_g(v) \tag{1}$$

where G is the group of maintainability guidelines from BCH (Table 1) 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 (Cruz et al. 2019). 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 Fig. 3. Thus, we can determine which patches had a positive, negative, or null impact on the project maintainability.

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 $\alpha = 0.05$ (Wilcoxon 1945). 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 (Pratt 1959). 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 (McGraw and Wong 1992).

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 4 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 1 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 ($\overline{x}$) 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.

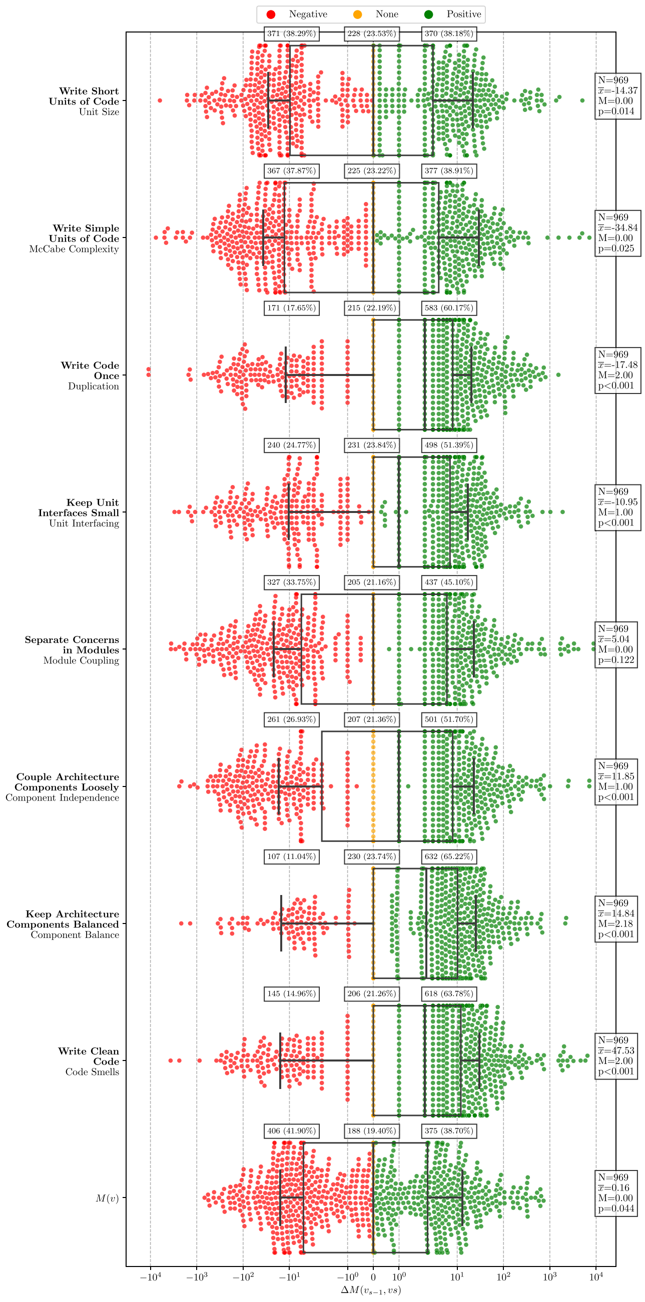


Fig. 4 Impact of the security patches per guideline and overall mean, $M(v)$. Each point of the plot represents the impact of a security patch on software maintainability. Red means the impact was negative, i.e., the patch harmed maintainability. Yellow means the patch did not have any kind of impact on maintainability. Green means the impact was positive, i.e., the patch improve software maintainability. For instance, in the *Write Short Units of Code* guideline, 38.29% of security patches harmed software maintainability; 23.53% of security patches had no impact on maintainability; and, 38.18% of security patches 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 vulnerabilities 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 (Zibran et al. 2011). Previous work showed a correlation between code smells and code duplication (Islam and Zibran 2016) 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 (Elkhail and Cerny 2019; Li and Paxson 2017) capable of harming the market value and economy of companies (Telang and Wattal 2007). 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 Fig. 4. 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. Fig. 4).

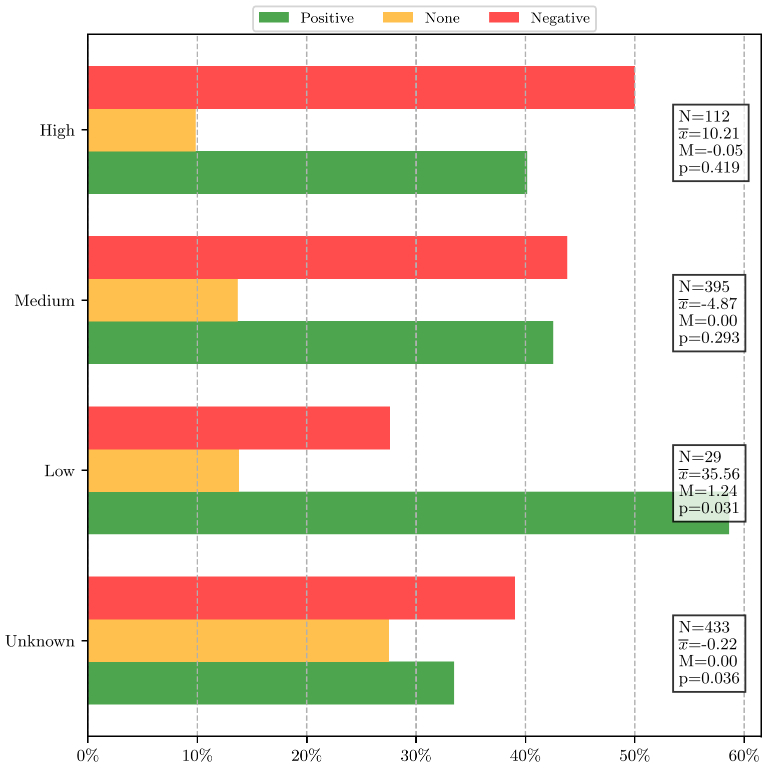


Fig. 5 Maintainability difference by vulnerability severity

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 5 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 (Fig. 6). 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

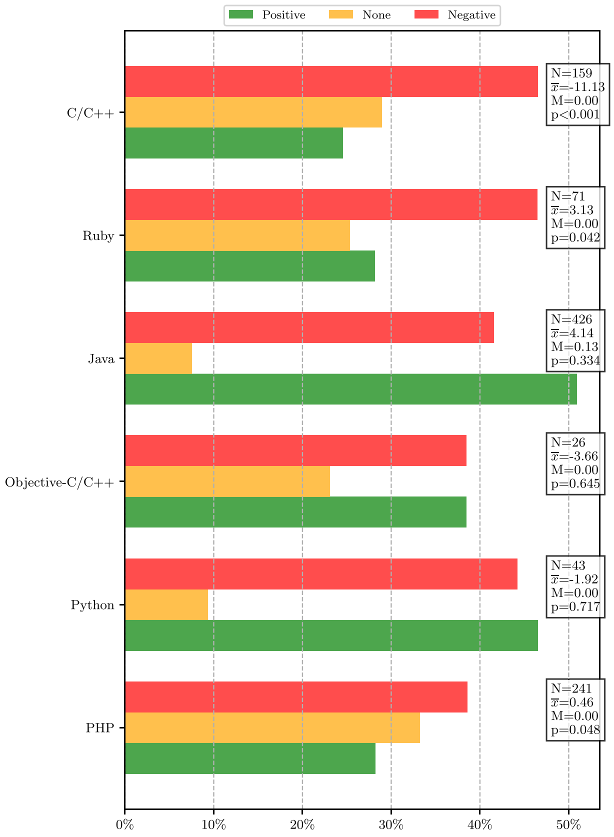
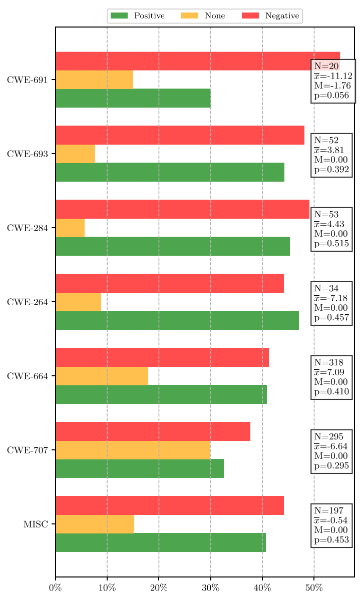


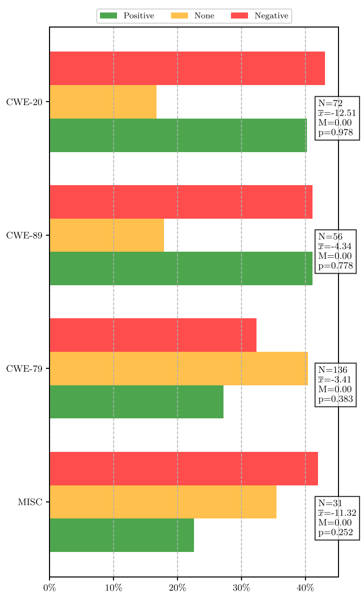
Fig. 6 Maintainability difference by programming language

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 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.

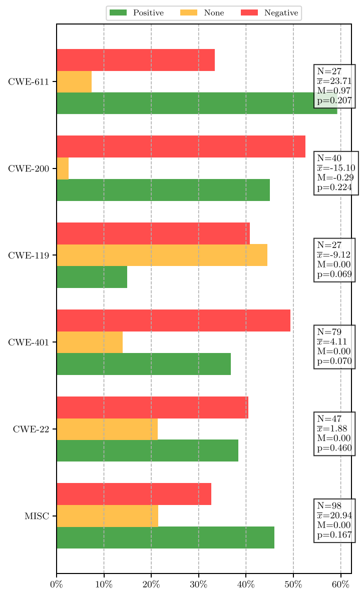
We expected the negative impact for programming languages on maintainability to be more severe, as arguably, poor design of programming languages for security and the lack of best practices application by developers lead to more buggy/vulnerable code (Ray et al. 2017; Berger et al. 2019). However, Fig. 6 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.



(a) Maintainability difference by first-level weaknesses from the *Research Concepts* list on Common Weakness Enumeration (CWE)



(b) Maintainability difference by sub-weaknesses of the *Improper Neutralization* Weakness (CWE-707)



(c) Maintainability difference by sub-weaknesses of the *Improper Control of a Resource Through its Lifetime* Weakness (CWE-664)

Fig. 7 Maintainability difference per weakness

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 Fig. 4, 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. Fig. 5). 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. Fig. 6).

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 2). Figure 7 shows three different charts. Figure 7-a, presents the impact of the 969 patches grouped by the first level weaknesses from the *Research Concepts*[12] list. While the Fig. 7-b and *-c* present the impact on maintainability for lower levels of weaknesses for the most prevalent weaknesses in Fig. 7-a: *Improper Neutralization* (CWE-707) and *Improper Control of a Resource Through its Lifetime* (CWE-664), respectively.

In Fig. 7-*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%) data points, respectively. Thus, we present an analysis of their sub-weaknesses on Fig. 7-*b* and -*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

[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

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 2—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 2).

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 Fig. 8. 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 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 ($p = 5.34\text{x}10^{-8}$).

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

[13]CVE-2016-0799 patch details available at https://github.com/openssl/openssl/commit/9cb177301fdab492e4cfef376b28339afe3ef663 (Accessed on September 20, 2021)

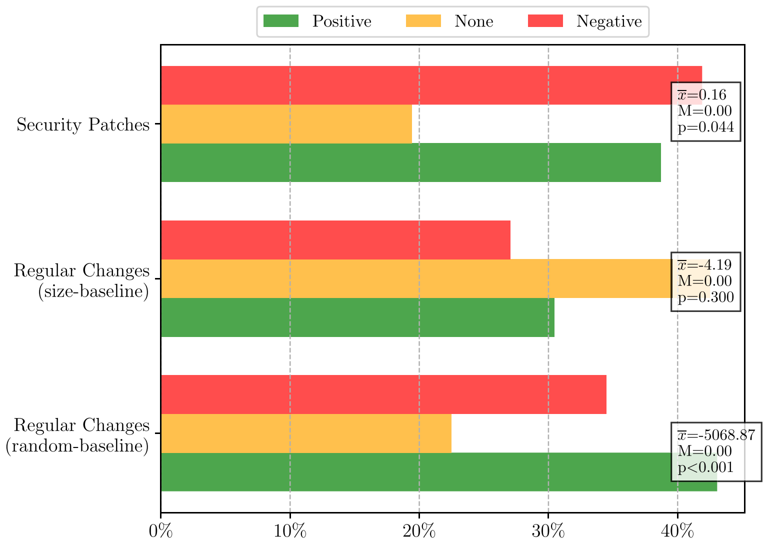


Fig. 8 Maintainability difference of security patches versus regular changes

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.

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 Fig. 4, 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 (Islam and Zibran 2016)—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 (Elkhail and Cerny 2019; Li and Paxson 2017). 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 (Li and Paxson 2017). 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 (Zazworka et al. 2011). 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 (Nistor et al. 2013; Kurilova et al. 2014; McGraw 2004). 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 information to security engineers on the vulnerabilities per se and the risk of their patches —alongside pull requests/code reviews. Static code analysis may be daunting due to the number of rules and different effects on maintainability. BCH claims to use the top-10 of the guidelines with the highest effects on maintainability. Yet, static analysis can be used to complement the BCH analysis by introducing the capability of vulnerability detection and support the prediction and prevention of the risk of a patch of hindering maintainability.

Computer Science Curricula Needs to be Updated: Computer Science curricula is not yet prepared to properly educate students for maintainable security. Students should be exposed to this problem to gain experience with it. Curricula should focus on the production of secure and maintainable code and alert to the 2 trade-off between both. These matters should be discussed and presented to students in software engineering courses. Universities should also encourage students to use code quality analysis tools such as BCH or similar ones. These tools have a great potential to make students aware of maintainability issues and beginner mistakes (e.g., coding practices violation).

Secure Programming Languages by Design: In general, our study shows that there is over 35% of patches with negative impact on software maintainability per programming language. Developers implementing code in `C/C++`, `Ruby` and `PHP` should pay extra care to their solutions/patches. In addition, programming languages should provide new design patterns to easily patch security weaknesses without endangering maintainability. Ultimately, new programming languages, both secure and maintainable by design, such as Wyvern (Nistor et al. 2013; Kurilova et al. 2014), should be designed to help developers be less vulnerability prone when writing and maintaining secure applications. One example is the need for designing new authorization mechanisms since these are one of the most complex security features to implement.

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 (Visser 2016; Baggen et al. 2012). 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 (Reis and Abreu 2017a) 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.

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 (Kataoka et al. 2002). In contrast to this work, Hegedűs et al. (2018) 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 (Li and Paxson 2017). 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. (Cruz et al. 2019) 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 (di Biase et al. 2019). 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 (Hegedűs et al. 2012). However, other studies show that the use of design patterns may introduce maintainability issues into software (Khomh and Gueheneuce 2008). 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 (Acar et al. 2017). 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 Ray et al. (2014, 2017). 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 Ray et al. (2014, 2017) 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.

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 (Olivari 2018; Cruz et al. 2019). 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.

Acknowledgements We thank SIG's Better Code Hub team for all the support as well as help in validating our methodology and results; and, Pedro Adão for the invaluable feedback in the early stages of the project.

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia with reference UIDB/50021/2020, a PhD scholarship (ref. SFRH/BD/143319/2019), the SecurityAware Project (ref. CMU/TIC/0064/2019)—also funded by the Carnegie Mellon Program—, and the FaultLocker Project (ref. PTDC/CCI-COM/29300/2017).

References

Acar Y, Stransky C, Wermke D, Weir C, Mazurek ML, Fahl S (2017) Developers need support, too: A survey of security advice for software developers. In: 2017 IEEE cybersecurity development (SecDev), pp 22–26. https://doi.org/10.1109/SecDev.2017.17

Alves TL, Correia JP, Visser J (2011) Benchmark-based aggregation of metrics to ratings. In: 2011 Joint conference of the 21st international workshop on software measurement and the 6th international conference on software process and product measurement, pp 20–29. https://doi.org/10.1109/IWSM-MENSURA.2011.15

Alves TL, Ypma C, Visser J (2010) Deriving metric thresholds from benchmark data. In: 2010 IEEE international conference on software maintenance, pp 1–10. https://doi.org/10.1109/ICSM.2010.5609747

Baggen R, Correia JP, Schill K, Visser J (2012) Standardized code quality benchmarking for improving software maintainability. Softw Qual J 20(2):287–307. https://doi.org/10.1007/s11219-011-9144-9

Berger ED, Hollenbeck C, Maj P, Vitek O, Vitek J (2019) On the impact of programming languages on code quality. arXiv:1901.10220

Bijlsma D, Ferreira MA, Luijten B, Visser J (2012) Faster issue resolution with higher technical quality of software. Softw Qual J. 20(2):265–285. https://doi.org/10.1007/s11219-011-9140-0

Chowdhury I, Zulkernine M (2010) Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities? In: Proceedings of the 2010 ACM symposium on applied computing, SAC ’10. pp 1963–1969, Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1774088.1774504

Common Criteria Working Group (2009) Common methodology for information technology security evaluation. Tech. rep., Technical report, Common Criteria Interpretation Management Board

Cruz L, Abreu R, Grundy J, Li L, Xia X (2019) Do energy-oriented changes hinder maintainability? In: 2019 IEEE International conference on software maintenance and evolution (ICSME), pp 29–40

di Biase M, Rastogi A, Bruntink M, van Deursen A (2019) The delta maintainability model: Measuring maintainability of fine-grained code changes. In: 2019 IEEE/ACM international conference on technical debt (TechDebt), pp 113–122

Elkhail AA, Cerny T (2019) On relating code smells to security vulnerabilities. In: 2019 IEEE 5th intl conference on big data security on cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE intl conference on intelligent data and security (IDS), pp 7–12

Foundation TO (2017) Owasp top 10 - 2017, The ten most critical web application security risks. Tech. rep., The OWASP Foundation. Release Candidate

Foundation TO (2017) Owasp top 10 - 2017, The ten most critical web application security risks. Tech. rep., The OWASP Foundation. Release Candidate

Hegedűs P, Kádár I, Ferenc R, Gyimóthy T (2018) Empirical evaluation of software maintainability based on a manually validated refactoring dataset. Inf Softw Technol 95:313–327. https://doi.org/10.1016/j.infsof.2017.11.012

Hegedűs P, Bán D, Ferenc R, Gyimóthy T (2012) Myth or reality? analyzing the effect of design patterns on software maintainability. In: Computer applications for software engineering, disaster recovery, and business continuity. Springer, Berlin, pp 138–145

Heitlager I, Kuipers T, Visser J (2007) A practical model for measuring maintainability. In: 6th International conference on the quality of information and communications technology (QUATIC 2007), pp 30–39. https://doi.org/10.1109/QUATIC.2007.8

International Organization for Standardization (2011) International standard ISO/IEC 25010 systems and software engineering - systems and software quality requirements and evaluation (SQuaRE) - system and software quality models

Islam MR, Zibran MF (2016) A comparative study on vulnerabilities in categories of clones and non-cloned code. In: 2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER), vol 3, pp 8–14

Just R, Jalali D, Inozemtseva L, Ernst MD, Holmes R, Fraser G (2014) Are mutants a valid substitute for real faults in software testing? In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering. ACM, pp 654–665

Kataoka Y, Imai T, Andou H, Fukaya T (2002) A quantitative evaluation of maintainability enhancement by refactoring. In: International conference on software maintenance, 2002. Proceedings., pp 576–585. https://doi.org/10.1109/ICSM.2002.1167822

Khomh F, Gueheneuce Y (2008) Do design patterns impact software quality positively? In: 2008 12th European conference on software maintenance and reengineering, pp 274–278. https://doi.org/10.1109/CSMR.2008.4493325

Kurilova D, Potanin A, Aldrich J (2014) Wyvern: Impacting software security via programming language design. In: Proceedings of the 5th workshop on evaluation and usability of programming languages and tools, pp 57–58

Li F, Paxson V (2017) A large-scale empirical study of security patches. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, CCS ’17, pp 2201–2215, Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3133956.3134072

Malavolta I, Verdecchia R, Filipovic B, Bruntink M, Lago P (2018) How maintainability issues of android apps evolve. In: 2018 IEEE international conference on software maintenance and evolution (ICSME), pp 334–344. https://doi.org/10.1109/ICSME.2018.00042

Maruyama K, Tokoda K (2008) Security-aware refactoring alerting its impact on code vulnerabilities. In: 2008 15th Asia-pacific software engineering conference, pp 445–452. https://doi.org/10.1109/APSEC.2008.57

McCabe TJ (1976) A complexity measure. IEEE Trans Softw Eng SE-2(4):308–320. https://doi.org/10.1109/TSE.1976.233837

McGraw G (2004) Software security. IEEE Secur Priv 2(2):80–83

McGraw KO, Wong SP (1992) A common language effect size statistic psychological bulletin. https://doi.org/10.1037/0033-2909.111.2.361

Nistor L, Kurilova D, Balzer S, Chung B, Potanin A, Aldrich J (2013) Wyvern: A simple, typed, and pure object-oriented language. In: Proceedings of the 5th Workshop on MechAnisms for SPEcialization, Generalization and InHerItance, MASPEGHI '13, pp 9–16, Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2489828.2489830

Olivari M (2018) Maintainable production: A model of developer productivity based on source code contributions. Master's thesis University of Amsterdam

Palomba F, Bavota G, Penta MD, Fasano F, Oliveto R, Lucia AD (2018) On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation. Empirical Softw Engg 23(3):1188–1221. https://doi.org/10.1007/s10664-017-9535-z

Ponta SE, Plate H, Sabetta A, Bezzi M, Dangremont C (2019) A manually-curated dataset of fixes to vulnerabilities of open-source software. In: Proceedings of the 16th international conference on mining software repositories, MSR '19. IEEE Press, p 383–387. https://doi.org/10.1109/MSR.2019.00064

Pothamsetty V (2005) Where security education is lacking. In: Proceedings of the 2Nd annual conference on information security curriculum development, InfoSecCD '05, pp 54–58, ACM, New York, NY, USA. https://doi.org/10.1145/1107622.1107635

Pratt JW (1959) Remarks on zeros and ties in the wilcoxon signed rank procedures. J Am Stat Assoc 54(287):655–667

Ray B, Posnett D, Devanbu P, Filkov V (2017) A large-scale study of programming languages and code quality in github. Commun ACM 60(10):91–100. https://doi.org/10.1145/3126905

Ray B, Posnett D, Filkov V, Devanbu P (2014) A large scale study of programming languages and code quality in Github. In: Proceedings of the 22Nd ACM SIGSOFT international symposium on foundations of software engineering, FSE 2014, 155–165, ACM, New York, NY, USA. https://doi.org/10.1145/2635868.2635922

Reis S, Abreu R (2017) A database of existing vulnerabilities to enable controlled testing studies. Int J Secur Softw Eng (IJSSE) 8(3). https://doi.org/10.4018/IJSSE.2017070101

Reis S, Abreu R (2017) Secbench: A database of real security vulnerabilities. In: Proceedings of the international workshop on secure software engineering in devops and agile development (SecSE 2017)

Schneier B (2006) Beyond fear: Thinking sensibly about security in an uncertain world. Berlin, Springer Science & Business Media

Shin Y, Meneely A, Williams L, Osborne JA (2010) Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Trans Softw Eng 37(6):772–787

Slaughter SA, Harter DE, Krishnan MS (1998) Evaluating the cost of software quality. Commun ACM 41(8):67–73

Telang R, Wattal S (2007) An empirical analysis of the impact of software vulnerability announcements on firm stock price. IEEE Trans Softw Eng 33(8):544–557

The OWASP Foundation (2009) OWASP application security verification standard 2009 - web application standard. Tech rep

Visser J (2016) Building maintainable software, java edition: Ten guidelines for future-proof code. O'Reilly Media, Inc

Visser J (2020) Sig/tUvit evaluation criteria trusted product maintainability: Guidance for producers. Available: https://bit.ly/3hnY0Am

Wilcoxon F (1945) Individual comparisons by ranking methods. Biometrics Bulletin 1(6):80–83

Xu H, Heijmans J, Visser J (2013) A practical model for rating software security. In: 2013 IEEE seventh international conference on software security and reliability companion, pp 231–232. https://doi.org/10.1109/SERE-C.2013.11

Zazworka N, Shaw MA, Shull F, Seaman C (2011) Investigating the impact of design debt on software quality. In: Proceedings of the 2nd workshop on managing technical debt, MTD '11, pp 17–23, Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1985362.1985366

Zibran MF, Saha RK, Asaduzzaman M, Roy CK (2011) Analyzing and forecasting near-miss clones in evolving software: An empirical study. In: 2011 16th IEEE international conference on engineering of complex computer systems, pp 295–304