Resolving Security Issues via Quality-Oriented Refactoring: A User Study

Domenico Gigante[1], Fabiano Pecorelli[25], Vita Santa Barletta[1], Andrea Janes[3], Valentina Lenarduzzi[4], Davide Taibi[24], Maria Teresa Baldassarre[1]
[1]University of Bari Aldo Moro — [2]Tampere University
[3]Vorarlberg University of Applied Sciences — [4]University of Oulu — [5]Jheronimus Academy of Data Engineering
domenico.gigante1@uniba.it; f.pecorelli@jads.nl; andrea.janes@fhv.at; valentina.lenarduzzi@oulu.fi; davide.taibi@oulu.fi; teresa.baldassarre@uniba.it

***Abstract*—Software quality is crucial in software development: if not addressed in early phases of the software development life cycle, it may even lead to technical bankruptcy, i.e., a situation in which modifications cost more than redeveloping the application from scratch. In addition, code security must also be addressed to reduce software vulnerabilities and to comply with legal requirements. In this work, we aim to investigate the relationship between refactoring code quality and software security, with the purpose of understanding whether and to what extent improving software quality could have a positive impact on software security as well. Specifically, we investigate to what extent rule violations of a software quality tool such as SonarQube overlap with rule violations of a software vulnerability tool like Fortify Static Code Analyzer. We first compared the rules encoded in the quality models of both tools, to discover possible overlapping cases. Later, we compared the issues raised by both tools on a set of open source Java projects; we also investigated the cases in which a quality refactoring process impacts over software security (thus removing one or more vulnerabilities). We furthermore validated our results statistically. Our results show that resolving software quality issues might also resolve security issues but only in part: many security issues still persist in the source code; also, some quality aspects are more likely to be improved in respect to others. In addition, this empirical study uncovers rule co-occurrences between the two tools. This study confirms the need for using a security-oriented static analysis tool to enforce software security instead of relying only on a quality-oriented one. Results have highlighted important insights for practitioners.**

I. Introduction

In an industrial software engineering context, quality is a factor that must be considered from different perspectives. For example, it is often an implicit requirement: stakeholders with little knowledge about software engineering assume that software is inherently reliable, secure, modifiable, adaptable, and so on, and therefore focus their attention on functional aspects and are surprised, if not unwilling, to pay for quality or to compromise on features in exchange for quality. In contrast, software failures, and in particular security vulnerabilities, can cause damage, lead to legal consequences, or have irreversible consequences for the reputation of the software producer.

Moreover, as popular project management frameworks suggest, spending time on quality implies spending less time on implementing features (assuming that the productivity of a team cannot be increased in the short run), or increasing costs (e.g., if we hire more engineers), or prolonging the time-to-market [1].

To mitigate the above-mentioned dilemma, industry has always been interested in reducing quality assurance costs, for example, as suggested by the Toyota Production System [2] and the subsequent Lean Thinking movement [3], through automated quality assurance mechanisms, in Japanese called *Jidoka*, in which "machines are able to detect the production of a single defective part and immediately stop themselves while asking for help." In software production, such automated quality assurance mechanisms are represented by software tools that are able to analyze the produced source code and warn developers of potential issues.

Various software tools to analyze software quality, and in particular code quality, exist. A popular category of these tools is represented by *static* analysis tools that, unlike *dynamic* tools, analyze source code without executing it. Executing source code requires reproducing the hardware context in which the source code is supposed to run, which can be non-trivial. For example, dynamically analyzing the software for an airport baggage handling system, in which custom-made hardware is also involved, is complex since many physical parts need to be simulated truthfully. Often, e.g., to determine performance issues or energy consumption, a dynamic analysis is necessary (e.g., as in [4], [5], [6]) but if feasible, analyzing source code statically is often preferred for cost reasons and sufficient to discover a variety of issues.

Therefore, the overall goal of software quality—regardless of how the code is analyzed—is to uncover potential quality problems in the source code. Even if software is developed using Agile methods, to allow changes that may even be late in the process [7], the development team has to consider an exponentially rising cost-of-change curve [8], i.e., it is always better to solve issues early in the process than later. Also from a technical debt point of view, it is in the interest of practitioners to maintain and improve code quality early in the process to reduce the risk of technical bankruptcy [9], i.e. that because of low quality, changes to the software take an unreasonable amount of time or that features cannot be

implemented without major redevelopments of large parts of the software.

For the above-mentioned reason, practitioners and researchers are aware of the benefits and need for code quality analysis tools and the adoption is increasing over time [10], [11]. Moreover, the impact, accuracy, and implications of using static analysis tools in a software development workflow are studied [12], [13]. Examples of such studies include comparisons of tools in terms of capability [14], detection agreement, and precision [15]. Other studies compared specific aspects detected by the tools, such as security [16] or concurrency defects [17], [18].

Two very popular static analysis tools are SonarQube[1] [10] and Fortify Static Code Analyzer[2]. SonarQube analyzes code quality based on a customizable quality model and applies the concept of technical debt to inform developers not only about code quality issues but also about the estimated (accumulated) cost of removing those issues.

Not intended as a static analysis tool for code quality in general, but specifically to analyze and test applications for security vulnerabilities, Fortify Static Code Analyzer (Fortify SCA), has been named for the 9th time one of the market leaders by Gartner in their application security testing market study, also in 2022 [19]. We found the term "Fortify SCA" or "Fortify Static" 54 times on IEEE Xplore searching within the full text of articles. We conjecture that this lower popularity in academic literature is not because security is less important than code quality but because Fortify SCA is only available with a commercial license, while SonarQube is available Open Source and because the topic of "security vulnerabilities" is more specific than "code quality".

The two tools discussed here—SonarQube and Fortify SCA—have been compared in several other papers, e.g., in [20] where the authors study the challenges in responding to alerts of static analysis tools, in [21], where the authors investigate what developers want and need from program analysis, or in [22] where the authors analyze to identify the limitations of static analysis security testing tools; we conclude from this even more that the research community considers these two products valid tools in the field of static source code analysis and worthwhile comparing their capabilities.

In many contexts, software security is a crucial aspect to consider during the software development process [23] in which developers aim to design the system to be resilient to external attacks [24]. Software vulnerabilities can cause loss of data, privilege escalation, race conditions, and other undesired effects that may affect the source code [25], [26].

The research community has been addressing the problem of vulnerabilities from different points of view, such as the impact on source code [27], [28], [29], or developing automated detection techniques [30].

Most of the approaches defined so far are based on source code and/or dynamic analysis [31], symbolic execution [32], and fuzz-testing [33], [34]. Some of them are also implemented within automated tools, such as Coverity Scan[3] or Fortify Static Code Analyzer.

Because of the importance of predicting, identifying, and resolving software vulnerabilities, researchers are also interested in correlating software vulnerabilities with other metrics. For example, Scandariato et al. [35], in study to which extent it is possible to use text mining methods to predict the same vulnerable software components as identified by Fortify SCA.

In order to address the aforementioned issues and to understand if enhancing the code quality also improves software security, we designed and conducted a user study investigating whether and to what extent improving software quality could have a positive impact on software security as well. To do so we use as a basis of our investigation two representative tools from both domains: SonarQube for the code quality domain and Fortify SCA for the software security domain.

The obtained results reveal findings interesting for researchers as well as practitioners: resolving software quality issues using SonarQube has an impact over software security as identified by Fortify SCA, but only to a limited extent. Many security issues still remain in the source code also after improvements. Moreover, some quality characteristics are more likely to be improved with respect to others. Regarding the tools' rules mapping we found new unexpected trends. Our study confirms the need for using a security-oriented Static Analysis Tool to enforce software security instead of relying only on a quality-based one.

Paper Structure: The study setting is described in Section II. The results are presented in Section III and discussed in Section IV. Section V identifies the threats to the validity of our study, while Section VI presents related works on static analysis tools. Finally, in Section VII we draw conclusions and provide an outlook on our future research agenda.

II. The Empirical Study

We designed our empirical study based on the guidelines defined by Wohlin et al. [36].

A. Goal and Research Questions

The goal of our study is to investigate the relationship between refactoring code quality and software security, with the purpose of understanding whether and to what extent improving software quality could have a positive impact on software security as well. The perspective is of researchers and practitioners, both interested in understanding whether these two aspects are related. The former could work on top of our research to build additional knowledge on explaining the reasons behind this relation, while the latter could be interested in applying such a process in their business to increase the efficiency of removing code vulnerabilities. To implement this goal, we define two Research Questions (RQs).

[1]https://www.sonarqube.org

[2]https://www.microfocus.com/en-us/cyberres/application-security/static-code-analyzer

[3]https://scan.coverity.com.

RQ$_1$. Do SonarQube rules overlap with Fortify SCA rules?

RQ$_2$. To what extent does software security improve by improving code quality?

First, we formulate a preliminary research question aiming at understanding if issues in the code raised by SonarQube rules are also considered security issues by Fortify SCA. In particular, considering that SonarQube classifies rules into three categories (Bug, Code Smell, and Vulnerability), we expect to find a perfect match of all the security vulnerability rules detected by SonarQube with those detected by Fortify SCA. However, considering that SonarQube rules are often misclassified [37], we also want to verify if the code affected by issues related to rules classified as Bug or Code Smell is also raising security issues from Fortify SCA.

Should this research question provide a negative answer, it does not make sense to address RQ2.

Once ensured that a possible relation between the two investigated aspects exists, we want to evaluate the extent to which improving software quality leads to improving software security as well. Thus, we define the above research question.

B. Context

The context of our empirical study consists of projects and static analysis tools. As already stated in the introduction, for the selection of automatic static analysis tools, we relied on two tools, based on their popularity in industry and academia: SonarQube for code quality assessment and Fortify SCA for what concerns security assessment. The reasons for this choice are further discussed in Section V.

We focused our study on a set of 23 projects written in Java, each of them satisfying the following criteria:

1) the project is hosted on GitHub (and publicly accessible);
2) its size is greater than 2.5KLOC (i.e., it contains more than 2500 Lines Of Code);
3) it has a SonarQube score worse than “A” for two or more characteristics (i.e., bugs, vulnerabilities, and code smells); and
4) it contains regression tests.

The ratio behind criteria (1) and (2) was to consider only real open-source projects with a non-trivial size. Criterion (3) aimed to discard projects with a good quality score in order to force the participants in our study to spend a significant amount of time to enhance the project’s static code quality. Finally, the last criterion allowed us to have sufficient confidence that each applied remediation did not introduce any error in the project logic while enhancing static code quality.

We focused only on Java projects in order to rely on one specific SonarQube quality profile, namely the set of rules checked by the tool. Indeed, by default, SonarQube provides different quality profiles for different programming languages. The same consideration stands for Fortify SCA: there are different sets or rules, each one focused on a different programming language, and just one group of "general rules" (language-agnostic), which was considered in this study.

Table I gives an overview of the final set of selected projects, the details, and URLs of the respective repositories can be found in the replication package (see Sect. II-D).

C. Data Collection & Analysis

In order to address our research questions, we first collected data from the subject repositories and tools and then we analyzed them. To this aim, we followed a two-steps approach. First, to address RQ$_1$, we performed a manual validation aiming to find correspondences between quality issues raised by SonarQube and security issues output by Fortify SCA. Then, to address RQ$_2$, we conducted a user study in which we asked participants to perform quality refactoring operations on the subject repositories and then we evaluated the impacts of such refactoring operations on security.

1) RQ$_1$ Manual mapping: The objective of RQ$_1$ was to find a correspondence between quality and security issues. To this aim we relied on the issues raised by the two aforementioned Static Analysis Tools: SonarQube for quality, and Fortify SCA for security issues.

As our study only focused on Java projects, we considered all the rules included in the SonarQube default Java quality profile, namely SonarWay[4]. This resulted in 627 SonarQube rules.

As for Fortify SCA, we selected the set of rules characterized by two possible values for "Code Language": "Universal" and "Java/JSP"[5], with a total number of 672; the rule pack used was version 2021.2.0.0008. Then, we compared SonarQube’s Java rules and Fortify SCA’s ones through a manual validation process. Specifically, for each of the 627 SonarQube rules, the first and the second author of the article separately read its description and tried to match it with one of the Fortify SCA’s rule. In case of a disagreement, the third author manually verified and took the final decision.

Finally, for each match found, we also annotated the suggested CWE both from SonarQube and Fortify SCA in order to verify the actual existence for a correspondence.

2) RQ$_2$ User Study: To address our second research question, we needed to collect data about the impacts of quality refactoring on software security. Hence, we designed a user study in which we asked participants to perform refactoring operations with the aim solving quality issues raised by SonarQube. Then we compared the issues generated by Fortify SCA before and after the quality refactoring to understand whether and to what extent there was an improvement in terms of security. We ran the study within the Computer Science program at the University of [Blind for the review]. All participants were third-year undergraduate students who were attending the “Software Quality” course. This course is composed by both face-to-face and laboratory lessons, and covered a wide set of topics: all aspects related to software quality (i.e., internal, external, and in-use); ISO standards

[4]https://docs.sonarqube.org/latest/instance-administration/quality-profiles

[5]https://vulncat.fortify.com/en/weakness?codelang=Java%2fJSP%3bUniversal&q=

TABLE I
PROJECTS' CHARACTERISTICS OVERVIEW

Name	Description	Size (KLOC)
nbvcxz	Password strength estimator	3.5
CalendarFX	Framework for creating sophisticated calendar user interfaces	33
openwayback	Engine to play back archived websites in the user's browser of the Internet archive Wayback Machine	45
jpmml-sklearn	Library and command-line application for converting Scikit-Learn pipelines to PMML	13
cache2k	In-memory high performance Caching library	24
Fling	Application to beam video files from computers to ChromeCast devices	3.6
scalable-coffee-shop	Demo implementation using event sourcing with an event-driven architecture, Apache Kafka and Java EE.	2.6
jnosql	Framework to create Jakarta EE applications with NoSQL	22
PDFrenderer	Library for rendering PDF documents to the screen using Java2D	20
mybatis-3	Framework making it easier to use a relational database with object-oriented applications	23
json-schema-validator	JSON schema validator	8.1
esapi-java-legacy	Web application security control library	19
db-scheduler	Persistent cluster-friendly scheduler	3.5
jackson-databind	General-purpose data-binding functionality and tree-model for the Jackson Data Processor	69
arquillian-core	Component model for integration tests, with dependency injection and container life cycle management	24
javaewah	A compressed alternative to the Java BitSet class	8.2
JSCover	JavaScript Code Coverage Tool that measures line, branch and function coverage	4.1
chromecast-java-api-v2	Java implementation of ChromeCast V2 protocol client	4.6
gchisto	A garbage collection log visualisation tool	8.4
Wikidata-Toolkit	Library to interact with Wikibase	21
initializr	Quickstart generator for Spring projects	16
DroidQuest	A Java recreation of the classic game Robot Odyssey	31
NFE	Electronic Invoice handler written in Java	88

related to software quality; software quality assessment, monitoring, and improvement processes and strategies [38]; and suggested tools for quality management (e.g., SonarQube), security management (e.g., Fortify SCA) and process control [39].

We initially recruited 128 participants, grouped in 58 groups of 2 or 3 people. At the time of our analysis, only 27 of these groups had completed all their tasks, hence our study discusses only the results of these first 27 groups.

Students were asked to perform their tasks by working in teams. Specifically, each student group performed the following steps on one of the projects reported in Table I:

1) Clone the project repository from GitHub to their local machine;
2) Run the code quality analysis by means of the SonarQube's cloud version (i.e., SonarCloud);
3) Run the code security analysis via Fortify SCA;
4) Annotate the list of issues presented by each tool, comprising, for each issue, file name and line;
5) Incrementally solve the issues output by SonarCloud, following the SCRUM methodology: in each "action plan" (equivalent to a SPRINT), a subgroup of issues was selected and solved. The quality refactoring operations were performed manually by the students, according to the issue and the correspondent solution provided by SonarCloud. At the end of each "action plan", another SonarCloud scan was run in order to check that each issue was actually removed; and
6) Run the code security analysis again via Fortify SCA once a quality target was reached.

The quality target varied according to the number of team components and the project's complexity which in most cases was: 0 bugs, 0 vulnerabilities, 0 code smells and a technical debt of 0 days. The exact numbers for all the projects are reported in the online appendix, see [6].

One observation needs to be made: in this study we needed to run Fortify SCA both before and after the refactoring operations. This was necessary because we needed a quantitative yardstick to compare how the project's security was affected by the quality refactoring.

We analyzed the resulting data by comparing the total number of occurring Fortify SCA issues before and after the quality refactoring was performed, both in absolute terms and percentages.

In case the number of Fortify SCA issues changed after the refactoring operations, a manual validation process was performed: one of the authors manually compared the affected source code with its own version before the refactoring; if the vulnerability was effectively removed (according to SonarQube suggestions) and Fortify SCA did not signal the issue anymore in the refactored source code, the refactoring was considered as valid and taken into consideration in the metrics count. This way, we did our best to exclude the presence of false negatives (i.e. genuine vulnerabilities not detected due to a random Fortify SCA check failure).

In order to mitigate false positive occurrences (i.e. pieces of code flagged as vulnerable but effectively not vulnerable), additional measures were taken: before and after the refactoring, an average of 45% of the total remaining issues - chosen by random sampling - was manually inspected for each project; if a false positive issue was discovered, it was suppressed (removed from the issues count on Fortify SCA).

After this filtering, we then computed three additional metrics, based on the rules mapping performed in RQ_1. In particular, we computed the "match count" as the number of issues identified by both SonarQube and Fortify SCA in

the initial snapshot; only rules with a match in the mapping performed in the context of RQ_1 were taken into consideration. Moreover, we measured the "solved count" as the number of matching issues that were spotted by Fortify SCA in the initial snapshot but were removed in the final snapshot, i.e., the number of Fortify SCA issues solved following to the quality refactoring operations. Again, only rules with a match in the mapping performed in the context of RQ_1 were taken into consideration. We calculated the "solved ratio" as the ratio between "solved count" and "match count".

Finally, we used statistical tests to confirm the variations in performance amongst the Fortify SCA issues distribution before and after the refactoring. To this aim, we relied on the paired Wilcoxon test [40]. At $alpha$=0.05, the results were supposed to be statistically significant. We also exploited Cliff's Delta (or d), a non-parametric effect size measure [41] for ordinal data, to assess the magnitude of the measured differences. To interpret the effect size values, we used well-established guidelines: negligible for $|d| < 0.10$, small for $|d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [41].

D. Replicability

To allow other researchers to replicate our study, we have published the complete raw data in the replication package [6]

III. Results

A. RQ_1: Do SonarQube rules overlap with Fortify SCA rules?

Table II reports the overall results regarding rule matches. As we can see, most of the SonarQube rules are related to design (i.e., code smells) or potential bugs. However, as one could expect, just a few portions of these rules find a match with Fortify SCA ones whose main focus is on security. Differently, for those rules aiming to identify vulnerabilities or security hotspots, we found a strong overlap between the two considered tools, namely 81% for vulnerabilities and 72% for security hotspots.

TABLE II
AGGREGATED ANALYSIS REGARDING THE RULES MAPPING. FOR EACH SONARQUBE CATEGORY, THE TABLE REPORTS THE NUMBER OF SONARQUBE RULES (#RULES), THE NUMBER OF RULES MATCHING WITH FORTIFY SCA RULES (#MATCHES), AND THE PERCENTAGE OF RULES MATCHING WITH FORTIFY SCA RULES (%MATCHES).

SQ categories	# rules	Fortify SCA	
		# matches	% matches
Bug	149	26	17%
Vulnerability	53	43	81%
Security Hotspot	36	26	72%
Code Smell	389	12	3%
Total	627	107	17%

The results suggest that SonarQube can contribute to a considerable extent (17%) in identifying and managing security-related issues. If we look at the other side of the coin, the results suggest that around 16% of Fortify SCA issues have a correspondence in SonarQube (i.e., 107 out of 672). Therefore, even if the results suggest that it is not possible to have significant coverage of Fortify SCA's rules, it seems that SonarQube could contribute by discovering a good percentage of issues of certain types. In the next research question, we analyze the magnitude of such a contribution.

[6]https://figshare.com/s/f229f6590c38032ebea3

TABLE III
NUMBER OF FORTIFY SCA ISSUES BEFORE AND AFTER THE REFACTORING, GROUPED BY SEVERITY.

Severities	initial snaphot	final snaphot	% solved
Critical	410	368	10%
High	3089	2651	14%
Medium	114	108	5%
Low	12052	9487	21%
Total	15665	12614	19%

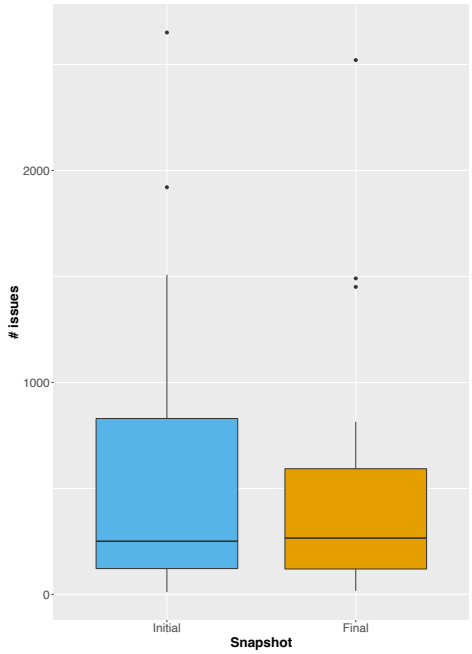

Fig. 1. Boxplots reporting the distribution of Fortify SCA issues over project in the initial (before refactoring) and the final (after refactoring) snapshots.

B. RQ_2: To what extent does software security improve by improving code quality?

Table III reports the number of Fortify SCA issues, grouped by severity levels, before and after the refactoring. Overall, by applying refactoring only relying on the output of SonarQube, leads to an improvement of 19% in terms of Fortify SCA solved issues. Inspecting the results by severities, the results report that SonarQube can help identify, and then solve, 10% of critical issues, 14% of high-severity issues, 5% of medium-severity issues, and 21% of low-severity issues.

Table IV reports the overall results regarding how projects' security was affected by implementing SonarQube's suggestions. Observing the table, in 18 out of the 27 cases projects

TABLE IV
OVERALL RESULTS REGARDING HOW PROJECTS' SECURITY IS IMPACTED BY REFACTORING BASED ON SONARQUBE'S SUGGESTIONS.

Group	Fortify SCA Total vulnerabilities (initial snapshot)	Fortify SCA Total vulnerabilities (final snapshot)	% Fortify SCA issues removed	Post-Refactoring situation	Match count	Solved count
G1	125	125	0	Equal	0	0
G2	317	296	7	Improved	64	64
G3	2651	2521	5	Improved	21	16
G4	37	24	35	Improved	2	2
G5	421	431	-2	Worsened	30	5
G6	251	272	-8	Worsened	6	6
G7	44	84	-91	Worsened	2	0
G8	149	154	-3	Worsened	0	0
G9	390	330	15	Improved	29	29
G10	613	661	0	Improved	10	5
G11	66	56	15	Improved	1	1
G12	1116	805	28	Improved	47	47
G13	1116	815	27	Improved	47	47
G14	119	69	42	Improved	0	0
G15	1047	802	23	Improved	69	57
G16	251	220	12	Improved	7	7
G17	610	575	6	Improved	43	13
G18	1508	276	82	Improved	19	19
G19	11	33	-200	Worsened	0	0
G20	227	170	25	Improved	0	0
G21	31	17	45	Improved	0	0
G22	87	115	-32	Worsened	0	0
G23	269	266	1	Improved	7	6
G24	215	218	-1	Worsened	0	0
G25	1921	1491	22	Improved	85	85
G26	149	150	-1	Worsened	0	0
G27	1494	1451	3	Improved	66	36

TABLE V
RESULTS FOR THE PAIRED WILCOXON'S SIGNED RANK TEST FOR STATISTICAL SIGNIFICANCE AND THE COHEN'S "D" EFFECT SIZE.

W	p-value	significance	d	Magnitude
283.5	0.006	***	0.416	Medium

got enhanced in security terms, 1 remained stable while 8 got a deterioration.

Columns "match count" and "solved count" of the table, report the number of matching Fortify SCA issues and solved matching Fortify SCA issues for each project, respectively.

As we can see, the maximum obtained enhancement is equal to 82% (for group G18) which corresponds to the solving of 1232 issues. Anyway, this happened just one time: in most of the cases, the obtained enhancement is between 0 and 45%, leaving a significant amount of issues in the source code.

On the other hand, when the number of Fortify SCA issues increases after refactoring (i.e., there is a decrease in security), there are a couple of concerning examples reporting a decrease of 91% (G7) or even 200% (G19). This could be related to more delicate refactoring operations that lead to introducing new security issues.

Overall, in most of cases, the vast majority of matching Fortify SCA issues are solved after applying the refactoring on top of SonarQube warnings. Moreover, we can observe that in 6 out of the 9 cases (i.e., 67%) in which there are no matching issues between Sonar and Fortify SCA, performing refactoring operations based on SonarQube warnings does not bring improvements in terms of security.

Figure 1 reports boxplots comparing the distributions of Fortify SCA issues over projects before and after the refactoring is performed. While the median values are almost identical, we can observe that the distribution in the final snapshot concentrates more towards lower values, thus indicating an overall improvement. To better investigate the significance of such a result, Table V reports the results of the paired Wilcoxon's signed rank test, as well as the Cohen's "d" effect size statistics. Results indicate that there is a significant improvement with respect to the number of security issues spotted by Fortify SCA, with a "medium" effect size.

Therefore, we can state that adopting SonarQube and refactoring the source code according to the warnings it generates, most likely leads to a significant improvement in terms of security aspects. However, it is still not possible to completely replace security-specific automatic static analysis tools (i.e., Fortify SCA in our case) since they can provide a more comprehensive overview of security aspects, identifying issues that cannot be identified in other ways.

IV. DISCUSSION

The achieved results revealed a number of insights that lead to implications for the software engineering community.

On the correspondence between SonarQube and Fortify SCA rules. We identified co-occurring rules that were triggered by both SonarQube and Fortify SCA on the same source code file line. Please note that these co-occurrences were not

TABLE VI
CO-OCCURRING RULES BETWEEN SONARQUBE AND FORTIFY SCA

SonarQube rule	Fortify SCA rule	# occurences
Delivering code in production with debug features activated is security-sensitive	System Information Leak	223
Unused method parameters should be removed	Dead Code: Unused Method	22
Null pointers should not be dereferenced	Redundant Null Check	14
Unused method parameters should be removed	Poor Error Handling: Overly Broad Throws	10
Standard outputs should not be used directly to log anything	Privacy Violation	9
Mutable fields should not be "public static"	Password Management: Hardcoded Password	8
Null pointers should not be dereferenced	Insecure Randomness	3
Null pointers should not be dereferenced	J2EE Bad Practices: Threads	1
Null pointers should not be dereferenced	Denial of Service: Parse Double	1
Silly equality checks should not be made	Code Correctness: Class Does Not Implement equals	1
Return values should not be ignored if they contain the operation status code	System Information Leak: Internal	1

expected since we did not include them in our initial mapping ($\mathbf{RQ}_1$). This was due to the fact that corresponding rules descriptions seem not to be similar enough. The co-occurring rules are presented in Table VI. As we can see, some of these trends are very rare to occur (just one occurrence for some of them), thus not necessarily indicating a real mapping between the two involved rules. However, in other cases, especially for the first row, it seems that there is a real association between the rules of the two tools.

By focusing on the descriptions of the two rules having 223 matching occurrences (i.e., row 1 of Table VI), even if both rules are abstract and do not address a specific problem (rather a general condition), we can see both have the word "debug" and one code example in common: the calling to printStackTrace() method over an Exception object; this seems to be fair and makes sense: so, this may be the case in which both the authors did not notice a valid association while carrying out the mapping.

In the remaining cases, the rules' descriptions do not show anything in common and, as such, may be random (or noisy) findings.

> 🔑 Even if some SonarQube and Fortify SCA rules are not directly associated, a high number of co-occurrences between them exist, thus indicating that further and larger investigations are needed to determine the actual set of corresponding rules.

Better together: One tool is not enough. According to the results obtained in our study, and specifically for $\mathbf{RQ}_1$, SonarQube is able to cover a small percentage (i.e., 16%) of the rules included in Fortify SCA. However, the most interesting finding has been obtained by looking at the severity of the overlapping rules. SonarQube allows to solve security issues classified with higher severity (High and Critical). Even if the improvement is not so significant, SonarQube can be adopted as an initial screening for security healthiness.

In light of these considerations, our results represent a call for further investigation regarding the role of static analysis tools for security issues detection. SonarQube and Fortify SCA classify similar rules differently or provide different classifications. It should be interesting to evaluate if the same observed trend can be recoverable with other static analysis tools, such as Coverity Scan.

> 🔑 SonarQube alone is not enough to provide significant support in discovering and removing security issues. However, it could be exploited to have a first preliminary overview of such issues.

V. THREATS TO VALIDITY

Construct Validity. First, in our study, a possible threat might be represented by the dataset used for the empirical investigation. We selected only Java projects to allow the rule set being consistent and we looked for repositories matching specific characteristics, as reported in Section II-B.

One could also argue about the selection of the quality-oriented and security-oriented Static Analysis Tools could have been used in this study, since many exist for both aspects. We chose to rely on SonarQube for quality assessment because it is one of the most used open-source quality-oriented Static Analysis Tools, as various studies demonstrate [42], [43]. Similarly, we selected Fortify SCA to assess projects' security since it is one of the "leaders" in application security testing according to the Gartner 2022 magic quadrants[7]. Additionally, we are aware that changing the group-repository assignments could have led to a different number of issues in the Fortify SCA final snapshots. Needless to say, also enterprise projects are prone to this risk: quality may depend on the developers' seniority, therefore, this may be considered an acceptable threat. Finally, it is well known that all static analysis tools are affected by false positive and negative identifications, an alert going away may simply mean that the refactoring has made the code too complicated for the tool to follow—which is why it is not reporting an alert anymore—or, conversely, the original alert may have been a false positive which is then hidden. In order to mitigate this problem, as described in section "Data Collection & Analysis", a manual inspection has been conducted before and after the refactoring operations.

Internal Validity. As for potential confounding factors that may have influenced our results, we are aware that some issues detected by SonarQube could be duplicated. Unfortunately, the

[7]https://www.gartner.com/en/documents/4001946

tool reports single issues violated in the same class multiple times. We mitigate this threat by manually searching for these cases and removing them.

External Validity.We are aware that different programming languages, and projects at different maturity levels could provide different results. This is why more experiments are needed on different projects, changing development technologies and sizes.

Conclusion Validity. In order to verify the significance of our results, in the context of RQ_2, we exploited appropriate statistical tests (i.e., Wilcoxon and Cliff's delta). Needless to say, although the security metrics are calculated with absolute care and attention, there is always a threat posed by manual computation.

VI. Related Work

The popularity of Static analysis tools adoption is increasing over the years creating a humus layer for the research. In this section, we report the relevant work on static analysis tools focusing on their usage [44], [45], [12], [18], rules and the detected problems [46].

The vast majority of the works investigated the detection capability of the available Static analysis tools [15] or their effective solving time [47] in different context [48]. A recent study [11] compared features and popularity of nine tools investigating also the empirical evidence on their validity. Results can help practitioners and developers to select the suitable tool against the other ones according to the measured information that satisfied better their needs. However, they did not evaluate their agreement and precision in the detection.

Other works focused on the performance Static analysis tools in term of accuracy in the issues detected and their effective [15] or if they are actually detecting issues related to some quality attributes [47], [45], [12] such as class fault- or change-proneness [18], [15]. Another important emerging usage is anomalies detection at different code level [49],

Even if some study demonstrated the effective in detecting some quality issues in the code [50], [12], others highlighted some critical issues regarding the detection models in term of rules classification and the rule severity assigned by the tool that can negative affect the detection accuracy [15] and the fixing estimated time [51], [52]. However, evaluating the performance results, they are discordant comparing different tools [15]. Several studies focused on the different rules provided by several tools (e.g. SonarQube, PMD, Jlint), and their results demonstrate some overlaps among the types of errors detected and no correlation among the rules [53], [15]. Each tool adopts different trade-offs to generate false positives and false negatives. Only one study [15] investigated the detection agreement and precision discovering little to no agreement among the tools and a low degree of precision. The issues related to the tool precision has been raised up also in another studies that highlighted the need to better clarify the precision of the tools. [54].

Looking at the developers' perception on the Static Analysis Tools usage, the average evaluation assessed the capability to find bugs [55]. However, developers are not sure about the usefulness of the rules [56], [57]. Commonly developers do pay attention to different rules categories, while they priority and remove violations only if they are related to rules with high severity [57] to avoid faults in their code [56]. An urgent need emerged from developers is the the need to reduce the number of detectable rules [58] or summarize them based on similarities [57]. Moreover, developers discovered that some tools do not capture all the possible defect even if they could be detected by the tools [59].

From a security perspective, code reviews with static analysis tools are recommended by several development processes to detect the security threats in software code [60]. It is necessary for software companies to integrate security within development processes by reducing code vulnerabilities and verify that legacy code is secure. Vulnerability is intended as one or more weaknesses that can be accidentally triggered or intentionally exploited and result in a violation of desired system properties [61]. Common Weakness Enumeration (CWE) is the most prominent effort in defining and classifying the security weakness in software. The CWE helps developers to describe and discuss software weaknesses in a common language and to evaluate coverage of tools targeting these weaknesses. Instead, NIST (National Institute of Standards and Technology) shifted its focus to determining what weaknesses existed in real software and could be found by tools [62]. Nine tools were run on the test suite and found that statistic analysis tools differed significantly with respect to precision and recall for different weakness. In addition, results showed that the sophisticated use of multiple tools would increase the rate of weakness detection and decrease the rate of false positives. Free and open source tools detected a minority of weaknesses only and using a security rule set could improve performance of the tools [63]. There is a little overlap among warnings from different tools and a meta-tool combining and cross-referencing output from multiple tools could be used to prioritize warnings [53].

VII. Conclusion

In this paper, we present an empirical study investigating the relation of quality refactoring on software security, with the purpose of understanding whether and to what extent improving software quality, based on the suggestions provided by quality-based SonarQube, could have a positive impact on software security, measured by means of Fortify SCA, as well. The main results of our investigation report that: (i) SonarQube allows to cover 17% of the rules checked by Fortify SCA; and (ii) the refactoring performed on top of SonarQube suggestions leads to a statistically significant improvement of 19% in terms of security issues raised by Fortify SCA.

To sum up, our article provides the following contributions:

1) A manual mapping of corresponding rules between SonarQube and Fortify SCA;
2) A user study that allows measuring the impacts of quality-based refactoring on security;

3) A series of implications for Industry, based on the results achieved;
4) A complete replication package [6] to allow other researchers reuse and replicate our analyses.

As future work, we plan to replicate our study on a larger dataset and in a different context. In particular, our idea is to rely on real usage data, and developers' refactoring, instead of basing our findings on a user study. To this aim, we plan to exploit the dataset by Nguyen et al. [64] which already provides such type of information. Additionally, we aim at investigating which SonarQube rules lead to specific refactoring operations that are likely to improve/worsen the security status of software systems.

REFERENCES

[1] H. M. Sneed, *Softwaremanagement*. Verlagsgesellschaft Rudolf Müller, 1987.

[2] T. Ōno, *Toyota Production System: Beyond Large-Scale Production*. Productivity Press, 1988.

[3] J. P. Womack and D. T. Jones, *Lean Thinking: Banish Waste and Create Wealth in Your Corporation*, 2nd ed. Free Press, 2003.

[4] A. Janes and B. Russo, "Automatic Performance Monitoring and Regression Testing During the Transition from Monolith to Microservices," in *International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2019, pp. 163–168.

[5] D. Gadler, M. Mairegger, A. Janes, and B. Russo, "Mining Logs to Model the Use of a System," in *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017, pp. 334–343.

[6] L. Corral, A. B. Georgiev, A. Janes, and S. Kofler, "Energy-Aware Performance Evaluation of Android Custom Kernels," in *International Workshop on Green and Sustainable Software*, 2015, p. 1–7.

[7] K. Beck and et al., "Manifesto for Agile Software Development," http://www.agilemanifesto.org/, 2001.

[8] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change*, 2nd ed. Addison-Wesley, 2004.

[9] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? Manage it? Ignore it? Software practitioners and technical debt," *Symposium on the Foundations of Software Engineering*, pp. 50–60, 2015.

[10] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, "How developers engage with static analysis tools in different contexts," *Empirical Software Engineering*, pp. 1–39, 2019.

[11] P. Avgeriou and et al., "An Overview and Comparison of Technical Debt Measurement Tools," *IEEE Software*, 2021.

[12] V. Lenarduzzi, N. Saarimäki, and D. Taibi, "Some SonarQube Issues have a Significant but SmallEffect on Faults and Changes. A large-scale empirical study," *Journal of Systems and Software*, vol. 170, 2020.

[13] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. Di Penta, "How open source projects use static code analysis tools in continuous integration pipelines," in *Int. Conf. on Mining Software Repositories*, 2017, pp. 334–344.

[14] M. Mantere, I. Uusitalo, and J. Roning, "Comparison of static code analysis tools," in *Int. Conf. on Emerging Security Information, Systems and Technologies*, 2009, pp. 15–22.

[15] V. Lenarduzzi, F. Pecorelli, N. Saarimaki, S. Lujan, and F. Palomba, "A critical comparison on six static analysis tools: Detection, agreement, and precision," *Journal of Systems and Software*, vol. 198, 2023.

[16] R. K. McLean, "Comparing static security analysis tools using open source software," in *Int. Conf. on Software Security and Reliability*, 2012, pp. 68–74.

[17] M. A. Al Mamun, A. Khanam, H. Grahn, and R. Feldt, "Comparing four static analysis tools for java concurrency bugs," in *Third Swedish Workshop on Multi-Core Computing (MCC-10)*, 2010.

[18] F. Pecorelli, S. Lujan, V. Lenarduzzi, F. Palomba, and A. De Lucia, "On the adequacy of static analysis warnings with respect to code smell prediction," *Empirical Software Engineering*, vol. 27, no. 3, 2022.

[19] D. Gardner, M. Horvath, and D. Zumerle, "Magic Quadrant for Application Security Testing," April 2022.

[20] N. Imtiaz, A. Rahman, E. Farhana, and L. Williams, "Challenges with Responding to Static Analysis Tool Alerts," in *International Conference on Mining Software Repositories (MSR)*, 2019, pp. 245–249.

[21] M. Christakis and C. Bird, "What developers want and need from program analysis: An empirical study," in *International Conference on Automated Software Engineering (ASE)*, 2016, pp. 332–343.

[22] I. Pashchenko, S. Dashevskyi, and F. Massacci, "Delta-Bench: Differential Benchmark for Static Analysis Security Testing Tools," in *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017, pp. 163–168.

[23] M. Dowd, J. McDonald, and J. Schuh, *The art of software security assessment: Identifying and preventing software vulnerabilities*. Pearson Education, 2006.

[24] G. McGraw, "Software security," *IEEE Security & Privacy*, vol. 2, no. 2, pp. 80–83, 2004.

[25] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *International Conference on Mining Software Repositories*, 2018, pp. 181–191.

[26] H. Plate, S. E. Ponta, and A. Sabetta, "Impact assessment for vulnerabilities in open-source software libraries," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 411–420.

[27] M. Finifter, D. Akhawe, and D. Wagner, "An empirical study of vulnerability rewards programs," in *Security Symposium*, 2013, pp. 273–288.

[28] S. Kim and H. Lee, "Software systems at risk: An empirical study of cloned vulnerabilities in practice," *Computers & Security*, vol. 77, pp. 720–736, 2018.

[29] D. Gonzalez, F. Alhenaki, and M. Mirakhorli, "Architectural security weaknesses in industrial control systems (ICS) an empirical study based on disclosed software vulnerabilities," in *International Conference on Software Architecture (ICSA)*, 2019, pp. 31–40.

[30] J. Svacina and et al., "On vulnerability and security log analysis: A systematic literature review on recent trends," in *International Conference on Research in Adaptive and Convergent Systems*, 2020, pp. 175–180.

[31] M.-T. Trinh, D.-H. Chu, and J. Jaffar, "S3: A symbolic string solver for vulnerability detection in web applications," in *Conference on Computer and Communications Security*, 2014, pp. 1232–1243.

[32] H. Li, T. Kim, M. Bat-Erdene, and H. Lee, "Software vulnerability detection using backward trace analysis and symbolic execution," in *International Conference on Availability, Reliability and Security*, 2013, pp. 446–454.

[33] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 259–269.

[34] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Symposium on Security and Privacy*. IEEE, 2010, pp. 497–512.

[35] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting Vulnerable Software Components via Text Mining," *Trans. Software Eng.*, vol. 40, no. 10, pp. 993–1006, 2014.

[36] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, and B. R. ¨and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. Springer, 2000.

[37] V. Lenarduzzi, N. Saarimäki, and D. Taibi, "Some sonarqube issues have a significant but small effect on faults and changes. a large-scale empirical study," *Journal of Systems and Software*, p. 110750, 2020.

[38] M. T. Baldassarre, M. Piattini, F. J. Pino, and G. Visaggio, "Comparing ISO/IEC 12207 and CMMI-DEV: Towards a mapping of ISO/IEC 15504-7," *International Conference on Software Engineering*, 2009.

[39] M. T. Baldassarre, D. Caivano, and G. Visaggio, "Software Renewal Projects Estimation Using Dynamic Calibration," *International Conference on Software Maintenance, ICSM*, pp. 105–115, 2003.

[40] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[41] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological bulletin*, vol. 114, no. 3, p. 494, 1993.

[42] D. Stefanović, D. Nikolić, D. Dakić, I. Spasojević, and S. Ristić, "Static code analysis tools: A systematic literature review," *International DAAAM Symposium*, vol. 31, no. 1, pp. 565–573, 2020.

[43] A. Nguyen-Duc and et al., "On the adoption of static analysis for software security assessment–A case study of an open-source e-government project," *Computers & Security*, vol. 111, p. 102470, 12 2021.

[44] N. Saarimäki, V. Lenarduzzi, and D. Taibi, "On the diffuseness of code technical debt in open source projects," in *International Conference on Technical Debt (TechDebt 2019)*, 2019.
[45] V. Lenarduzzi, A. Martini, D. Taibi, and D. A. Tamburri, "Towards Surgically-precise Technical Debt Estimation: Early Results and Research Roadmap," in *International Workshop on Machine Learning Techniques for Software Quality Evaluation*, 2019, pp. 37–42.
[46] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software," in *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 470–481.
[47] D. Marcilio, R. Bonifácio, E. Monteiro, E. Canedo, W. Luz, and G. Pinto, "Are Static Analysis Violations Really Fixed? A Closer Look at Realistic Usage of SonarQube," in *International Conference on Program Comprehension*, 2019, p. 209–219.
[48] B. Lu, W. Dong, L. Yin, and L. Zhang, "Evaluating and Integrating Diverse Bug Finders for Effective Program Analysis," in *Software Analysis, Testing, and Evolution*, 2018, pp. 51–67.
[49] P. Tomas, M. J. Escalona, and M. Mejias, "Open source tools for measuring the Internal Quality of Java software products. A survey," *Computer Standards and Interfaces*, vol. 36, no. 1, pp. 244–255, 2013.
[50] V. Lenarduzzi, F. Lomio, H. Huttunen, and D. Taibi, "Are SonarQube Rules Inducing Bugs?" *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019.
[51] M. T. Baldassarre, V. Lenarduzzi, S. Romano, and N. Saarimäki, "On the diffuseness of technical debt items and accuracy of remediation time when using SonarQube," *Information and Software Technology*, vol. 128, p. 106377, 2020.
[52] V. Lenarduzzi, V. Mandić, A. Katin, and D. Taibi, "How Long Do Junior Developers Take to Remove Technical Debt Items?" in *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020.
[53] N. Rutar, C. B. Almazan, and J. S. Foster, "A comparison of bug finding tools for Java," in *Symposium on Software Reliability Engineering*, 2004, pp. 245–256.
[54] G. Liang, L. Wu, Q. Wu, Q. Wang, T. Xie, and H. Mei, "Automatic construction of an effective training set for prioritizing static analysis warnings," in *International Conference on Automated software engineering*, 2010, pp. 93–102.
[55] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.
[56] D. Taibi, A. Janes, and V. Lenarduzzi, "How developers perceive smells in source code: A replicated study," *Information and Software Technology*, vol. 92, pp. 223–235, 2017.
[57] C. Vassallo, S. Panichella, F. Palomba, S. Proksc, H. C. Gall, and A. Zaidman, "How developers engage with static analysis tools in different contexts," *Empirical Software Engineering*, 2019.
[58] T. Muske, R. Talluri, and A. Serebrenik, "Repositioning of Static Analysis Alarms," in *International Symposium on Software Testing and Analysis*, 2018, p. 187–197.
[59] F. Thung, Lucia, D. Lo, L. Jiang, F. Rahman, and P. T. Devanbu, "To What Extent Could We Detect Field Defects? An Extended Empirical Study of False Negatives in Static Bug-Finding Tools," *Automated Software Engineering*, vol. 22, no. 4, p. 561–602, 2015.
[60] D. Baca, K. Petersen, B. Carlsson, and L. Lundberg, "Static Code Analysis to Detect Software Security Vulnerabilities - Does Experience Matter?" in *International Conference on Availability, Reliability and Security*, 2009, pp. 804–810.
[61] M. T. Baldassarre, V. S. Barletta, D. Caivano, and M. Scalera, "Integrating Security and Privacy in Software Development," *Software Quality Journal*, vol. 28, no. 3, p. 987–1018, sep 2020.
[62] A. Delaitre, B. Stivalet, P. Black, V. Okun, T. Cohen, and A. Ribeiro, "SATE V Report: Ten Years of Static Analysis Tool Expositions," 2018-10-23 2018.
[63] A. Wagner and J. Sametinger, "Using the Juliet Test Suite to compare static security scanners," in *International Conference on Security and Cryptography (SECRYPT)*, 2014, pp. 1–9.
[64] H. Nguyen, F. Lomio, F. Pecorelli, and V. Lenarduzzi, "PANDORA: Continuous mining software repository and dataset generation," in *International Conference on Software Analysis, Evolution and Reengineering (SANER2022)*, 2022.