Noname manuscript No.
(will be inserted by the editor)

Rubbing Salt in The Wound? A Large-Scale Investigation into The Effects of Refactoring on Security

Emanuele Iannone · Zadia Codabux · Valentina Lenarduzzi · Andrea De Lucia · Fabio Palomba

Received: date / Accepted: date

Abstract Software refactoring is a behavior-preserving activity to improve the source code quality without changing its external behavior. Unfortunately, it is often a manual and error-prone task that may induce regressions in the source code. Researchers have provided initial compelling evidence of the relation between refactoring and defects, yet little is known about how much it may impact software security. This paper bridges this knowledge gap by presenting a large-scale empirical investigation into the effects of refactoring on the security profile of applications. We conduct a three-level mining software repository study to establish the impact of 14 refactoring types on (i) security-related metrics, (ii) security technical debt, and (iii) the introduction of known vulnerabilities. The study covers 39 projects and a total amount of 7,708 refactoring commits. The key results show that refactoring has a limited connection to security. However, *Inline Method* and *Extract Interface* statistically contribute to improving some security aspects connected to encapsulating security-critical code components. *Extract Superclass* and *Pull Up Attribute* refactoring are commonly found in commits violating specific security best practices for writing secure code. Finally, *Extract Superclass* and *Extract & Move Method* refactoring tend to occur more often in commits contributing to the introduction of vulnerabilities. We conclude by distilling lessons learned and recommendations for researchers and practitioners.

Keywords Refactoring; Software Security; Empirical SE.

Emanuele Iannone, Andrea De Lucia, Fabio Palomba
SeSa Lab — University of Salerno, Italy
E-mail: eiannone@unisa.it, adelucia@unisa.it, fpalomba@unisa.it

Zadia Codabux
University of Saskatchewan, Canada
E-mail: zadiacodabux@ieee.org

Valentina Lenarduzzi
University of Oulu, Finland
E-mail: valentina.lenarduzzi@oulu.fi

1 Introduction

In 1999, Fowler defined the term *"software refactoring"* to indicate the activities developers perform to improve the internal structure of source code without changing its external behavior [1]. Since then, the research community has investigated refactoring from multiple perspectives [2, 3, 4, 5], proposed novel recommendation systems to help developers refactor source code [6, 7, 8], empirically investigated why developers refactor source code [9, 10], studied the current barriers preventing refactoring in practice [11, 12, 13, 14], and explored the effects of refactoring on source code dependability [15, 16, 17, 18].

One of the most worrisome results of these empirical analyses is that refactoring might induce defects. Bavota et al. [19] and, more recently, Di Penta et al. [16] have indeed shown that refactoring operations can induce faults in a non-negligible number of cases—this is likely due to refactoring operations done manually rather than supported by semi-automated tools [12].

This study builds on this line of research and investigates the relationship between refactoring and software security, defined as the property that allows the software to continue working correctly under potential risks due to external malicious attacks that may cause loss or harm [20]. Our study is based on the assumption that refactoring operations performed by developers can lead to variations in the security level of an application. In the first place, this assumption is justified by early work studying the relation between refactoring and security measured in various ways. In particular, Abid et al. [15] recently proposed a search-based security-aware refactoring recommender that suggests the refactoring operations to apply to obtain the best trade-off between code maintainability and security degradation. While the main focus of such work was the definition of a novel refactoring recommender, they also conducted a preliminary motivational analysis to correlate (i) the presence of 14 automatically-detectable refactoring types [21] and (ii) the QMOOD metrics [22] with eight data-access security indicators proposed in literature [23]. The analysis considered a single snapshot of 30 open-source software systems and revealed that some refactoring types are negatively correlated to security—i.e., their application caused the worsening of certain security characteristics. Among their findings, they observed a negative correlation between the application of *Extract Superclass* refactoring operation [1] and data-access security indicators—which is something we also observed in the context of our research. For instance, let us consider the case of project Conversations[1] at the revision `2067b9bd`, where the application of an *Extract Superclass* has led to the extraction of class `XmppUri` from class `Invite`. This refactoring caused the introduction of new security-sensitive attributes. Thus, it seems reasonable to believe that refactoring might impact an application's security profile. Indeed, *Extract Superclass* revolves around modifying hierarchies to create a common superclass for a set of classes. By design, a superclass is more accessible than subclasses, which might expose previously hidden sensitive parts of

[1] `https://github.com/iNPUTmice/Conversations`

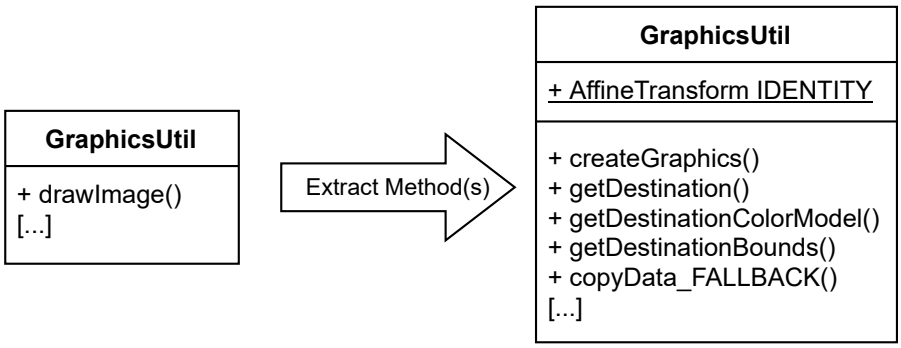

Fig. 1: Simplified graphical representation of the changes applied to `GraphicsUtil` class at the revision `8309088a` in project BATIK.

the program, increasing the chances of introducing vulnerabilities. In a similar manner, Abid et al. [15] found correlations between other refactoring operations (e.g., *Move Method* [1]) and other security-related aspects.

Our research identifies a set of refactoring types whose application might actually lead to variations of the security level of the code being refactored—as detailed later in Section 2.3.1. In this sense, we build our empirical analysis upon logical reasoning, through which we hypothesized and verified the extent of the identified relations. For example, we hypothesize that the *Pull Up Attribute* refactoring [1]—i.e., moving an attribute from a subclass to a superclass, changing its visibility and external accessibility—potentially leads to security drifts due to the wider exposition of the attribute. As an additional example, let us consider a case observed in the context of our research. This pertains to project BATIK[2] at the revision `8309088a`. The commit applied a great restructuring of the classes, as also pointed out by the commit message reported in the following:

> *"The deepest architectural change is a strong move towards tiling everything [...]".*

Elaborating on the modifications performed in this commit, the `GraphicsUtils` class was affected by the various changes, being subject to several *Extract Method* refactoring operations. Specifically, the main restructuring involved the long method `drawImage()`, whose logic was decomposed into several smaller and more cohesive methods. In the version before the change, the `drawImage()` method allocated a new instance of the `AffineTransform` class each time it was called. Likely, this was judged as invalid, so making the commit's author introduce a new class variable (i.e., a `static` class field) pointing to an instance of `AffineTransform` class named `IDENTITY` having `public` visibility. However, they likely ended up leaving it not-`final`, making it modifiable from any other class that has access to `GraphicsUtils`—i.e.,

[2] `https://github.com/apache/batik`

potentially any project that includes BATIK library in their classpath. This scenario represents a pointless exposure of the class variable to any change, likely introducing bugs or even leaking information that should not be disclosed to clients. Figure 1 shows the `GraphicsUtils` class before and after the application of the multiple *Extract Method* refactoring operations.

Based on these observations and recognizing the significant research advances done by Abid et al. [15], we aim to substantially enlarge the knowledge of the relation between refactoring and security, using different statistical methods and looking at different aspects characterizing software security. More specifically, we aimed at defining a *theory* that could provide quantitative indications of how different refactoring operations may impact security under different perspectives. Hence, we consider the change history information of 39 software projects and conduct a three-level quantitative analysis. We first assess the extent to which 14 refactoring types extracted by REFACTORINGMINER [24] may affect the source code from a security perspective. As such, we measure the effects of refactoring on (i) a set of security metrics available in literature [23] and computed with a homemade tool, coined SURFACE (**S**ec**UR**ity **F**l**A**ws metri**C**s **E**xtractor), that we publicly release to the research community, and (ii) security-related technical debt, as computed by SONARQUBE.[3] In doing so, we use similar statistical instruments as in previous studies investigating the relation between refactoring and source code quality [9], program comprehension [25], and defects [16]. In particular, statistical models specify mathematical relationships between one or more independent variables (in our case, the refactoring operations and a set of confounding variables) and dependent variables (in our case, the source code security level computed using security metrics and technical debt). As such, statistical modeling allows us to formally represent our theory [26], perfectly fitting the goals of our study. As the last part of the study, we verified how refactoring could lead to the introduction of known vulnerabilities mined from the National Vulnerability Database (NVD) [27]. In this case, we first measured the number of times refactoring operations are performed in commits where known vulnerabilities are introduced; then, we conducted a finer-grained manual investigation to understand the extent to which refactoring operations are actually contributing to the introduction of vulnerabilities.

The key results of our investigation show a limited connection between refactoring and security. Indeed, we discover that most of the refactoring operations do not have a significant impact on any of the security perspectives considered. At the same time, we highlight some noticeable exceptions: *Inline Method* and *Extract Interface* are the refactoring operations that appear to be statistically significant when it turns to the improvement of some security aspects connected to encapsulation, while *Extract Superclass*, and *Pull Up Attribute* are linked to an increase of violations to certain security practices to write secure code. Furthermore, the *Extract Superclass* and *Extract & Move Method* refactoring types tend to occur more often in commits contributing to

[3] Link: `https://www.sonarqube.org`

the introduction of real vulnerabilities. Based on our findings, we identify and distill a set of concrete issues and challenges that the refactoring community should face to better support developers. To sum up, this study provides the following contributions:

1. An evidence-based investigation into the relation between refactoring and security that targets the problem under three different perspectives, such as security metrics, security-related technical debt, and known vulnerabilities;
2. A research roadmap that researchers in the field can exploit to understand further the circumstances that lead refactoring to negatively affect security and provide automated support for practitioners;
3. An online appendix [28] reporting all the data and scripts used in the study to allow researchers to replicate and conduct additional investigations.

Structure of the paper. Section 2 reports the methodology employed in the study, while Section 3 discusses the results achieved. In Section 4, we provide an overview of the main discussion points and implications of the results for the research community and practitioners. Section 5 reports on the threats that may have biased our findings. Section 6 discusses the related literature. Finally, Section 7 concludes the paper.

2 Research Methodology

The *goal* of this study is to assess the relation between refactoring and security, with the *purpose* of understanding how refactoring operations applied by developers introduce security threats. The *perspective* is of both researchers and practitioners: the former are interested in understanding which additional support developers would require when performing refactoring; the latter are interested in evaluating the potential consequences of refactoring on source code dependability. Our study was designed based on the guidelines proposed by Runeson and Host [29] and follows the ACM/SIGSOFT Empirical Standards recently introduced and discussed by Ralph et al. [30].[4]

2.1 Research Questions and Methodological Overview

The empirical study is based on three levels of analysis. Following the preliminary investigation by Abid et al. [15]—who observed a correlation between the amount of refactoring operations applied and security metrics—we aimed at assessing the security implications of refactoring operations on security indicators in an effort to provide insights into the potential compromise a developer should pay attention to while improving source code quality.

[4] Given our study and currently available standards, we followed the general guidelines when reporting the study design and results.

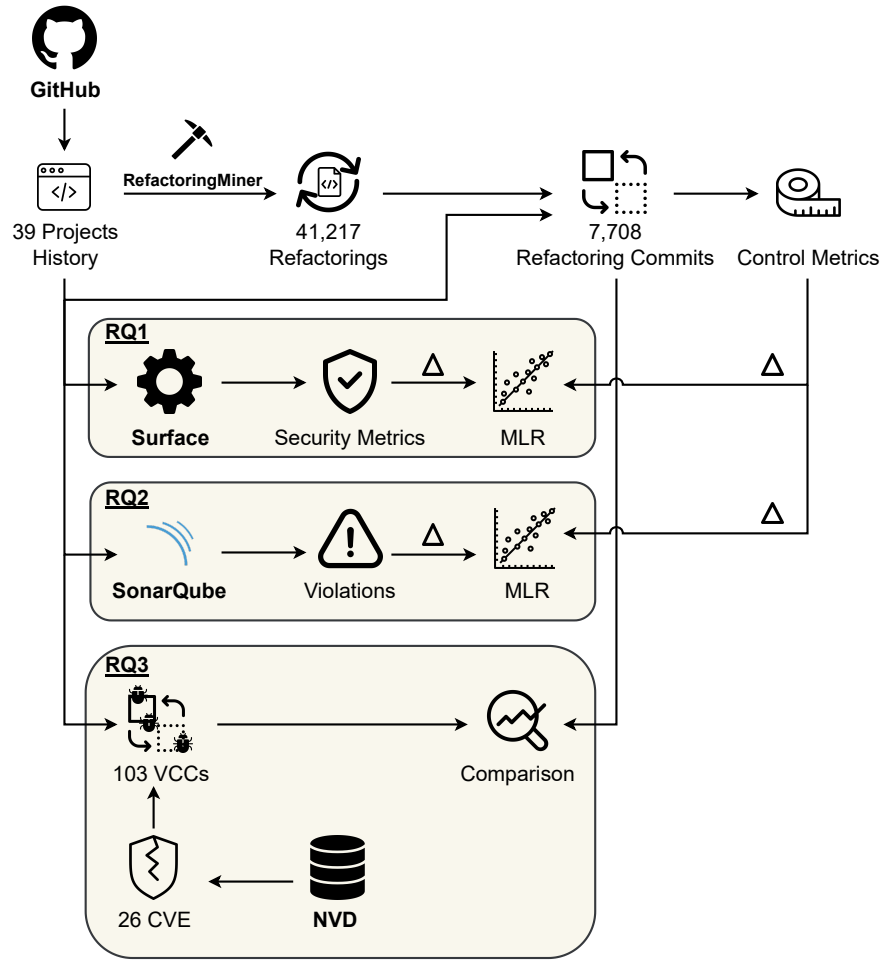

Fig. 2: Methodological steps employed in our study.

We start facing this research objective using two analyses: the first focused on security-related metrics that indicate portions of source code whose characteristics may lead the code to be more exposed to security risks [23]; the second targeting technical debt [31] that highlights the design and implementation issues that might represent exploitable security flaws. These two analyses are by nature complementary: security-related metrics focus on weak constructs implemented in the source code, while security technical debt measures on higher-level poor design or implementation solutions that might possibly impact the security profile of an application. As further explained in Section 2.3, we conducted these analyses by measuring developers' activities, and their implications for source code security by running tools and analyses on commits

where refactoring has been applied. These goals led to the formulation of the following two research questions:

RQ$_1$. *To what extent do refactoring operations impact security metrics?*

RQ$_2$. *To what extent do refactoring operations impact security-related technical debt?*

While the first two research questions allowed us to uncover possible negative effects given the application of refactoring operations on security, these were not sufficient nor comprehensive. Both security-related metrics and technical debt focus on *potential* risks for source code security; yet, this does not still clarify if and how refactoring has an impact on the introduction of *real* security threats. For this reason, we continued our empirical investigation by assessing how the application of refactoring operations over the change history of software projects leads to the introduction of known software vulnerabilities. This reasoning led to our last research question:

RQ$_3$. *To what extent do refactoring commits contribute to the introduction of real software vulnerabilities?*

The study can be configured as a quantitative investigation [32] where we seek to find statistically significant findings from a large amount of data. While the next sections detail the data collection and analysis procedures used to address our research questions, Figure 2 overviews the methodology employed in our study. In short, when addressing **RQ$_1$** and **RQ$_2$** we first run three tools, namely an automated refactoring detector called RefactoringMiner [24], a security metric tool named Surface, and a static code analyzer called SonarQube over all the commits of the considered projects. Afterward, we use the data collected to compute the difference in terms of security metrics and debt between the refactoring commits and their predecessors to indicate how the refactoring operations have changed these measures. Finally, the variation of security metrics and debt were used as dependent variables of Multinomial Log-Linear regression models [33] that allowed us to identify which refactoring operations are statistically related to their increase or decrease while controlling for factors like complexity, lines of code, and code churn.

As for **RQ$_3$**, we mined the vulnerability-fixing commits of known vulnerabilities affecting the software projects considered in our study and available on a public dataset of vulnerabilities, namely, the National Vulnerability Database (NVD) [27]. Then, we employed an automated mechanism based on the SZZ algorithm [34] to identify the commits responsible for the introduction of those known vulnerabilities and combined this information with the one from RefactoringMiner to obtain the number of times refactoring operations

Table 1: Summary of the considered software projects. The last column 'NVD' indicates whether the corresponding project has known vulnerability data.

Project	#Commits	#Ref.Commits	#Refact.	NVD
ARCHIVA	4,741	363	2,399	✖
BATIK	2,196	197	1,747	✖
CAYENNE	1,269	99	384	✖
COCOON	10,334	559	2,652	✖
COMMONS-BCEL	1,324	37	959	✖
COMMONS-BEANUTILS	1,209	41	385	✖
COMMONS-CLI	855	26	115	✖
COMMONS-CODEC	1,732	53	363	✖
COMMONS-COLLECTIONS	2,893	145	1,633	✖
COMMONS-CONFIGURATION	2,930	250	1,293	✖
COMMONS-DAEMON	982	2	3	✖
COMMONS-DBUTILS	603	13	66	✖
COMMONS-DIGESTER	2,143	47	373	✖
COMMONS-EXEC	616	14	45	✖
COMMONS-FILEUPLOAD	914	15	104	✖
COMMONS-IO	2,055	88	329	✖
COMMONS-JELLY	1,938	73	203	✖
COMMONS-JEXL	1,533	101	657	✖
COMMONS-JXPATH	597	58	436	✖
COMMONS-NET	2,100	63	301	✖
COMMONS-OGNL	607	18	298	✖
COMMONS-VALIDATOR	1,339	42	150	✖
COMMONS-VFS	2,080	133	771	✖
FELIX	3,489	247	2,173	✖
HIVE	5,919	892	5,175	✖
HTTPCOMPONENTS-CLIENT	2,714	278	2,822	✖
HTTPCOMPONENTS-CORE	2,760	348	2,967	✖
SANTUARIO-JAVA	2,755	158	1,038	✖
THRIFT	2,912	33	187	✖
ZOOKEEPER	1,487	159	979	✖
CONVERSATIONS	6,426	566	1,284	✔
CANDLEPIN	10,967	696	3,597	✔
HAWTIO	8,856	144	521	✔
JBOSS-NEGOTIATION	307	20	120	✔
JENKINS	30,632	1,315	3,417	✔
JOLOKIA	1,695	196	607	✔
JUNRAR	233	20	141	✔
LITEMALL	1,093	37	135	✔
STRUTS1-FOREVER	4,255	163	571	✔
Overall	**133,490**	**7,708**	**41,217**	-

"#Ref.Commits." refers to the number of commits having refactorings
"#Refact." refers to the number of refactoring instances observed

are likely to have contributed to a known vulnerability. A manual qualitative investigation later contextualized the statistical analyses to understand further and discuss the quantitative results. The detailed methodological steps adopted to collect the described data are reported in Section 2.3.

2.2 Context of the Study

The *context* of the study was composed of open-source software projects and, in particular, their change history information. In this respect, we exploited the *Technical Debt Dataset* [35], which is a curated collection of data coming from 39 Java projects mainly from the Apache Software Foundation ecosystem. Despite belonging to a single ecosystem, the majority of such projects were originally selected by following the diversity guidelines introduced by Nagappan et al. [36], i.e., they were selected by addressing the representativeness of projects in terms of age, size, and domain, and the Patton's "criterion sampling" [37], namely, they are more than four years old, have more than 200 commits and 100 classes, and have more than 100 issues reported in their issue tracking system. As such, this dataset minimizes by design possible threats to external validity. To further verify the properties of this dataset, we have manually investigated the corresponding Github repositories and discovered that all of them adhere to a strict code of conduct [38] and regularly review source code to improve their quality processes [39]. This analysis further confirmed the suitability of the dataset. It is important to note that nine of the considered systems also appear in the *National Vulnerability Database* (NVD) [27], which was initially developed by the U.S. NIST Computer Security Division [40] to collect and provide public information about known vulnerabilities affecting software systems and their causes. Such a database includes a comprehensive set of publicly known vulnerabilities: each of them is described through CVE (Common Vulnerabilities and Exposure [41]) records and is enriched with additional pieces of information such as external references, severity (computed using the Common Vulnerability Scoring System - CVSS), the related weakness type (Common Weak Enumeration - CWE), and the known affected software configurations (Common Platform Enumerations - CPEs). NVD aggregates information from multiple data sources and is widely considered a reliable data source [42, 43, 44].

While all the projects were considered when addressing $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$, only the nine systems overlapping with the NVD could be used for $\mathbf{RQ}_3$ as these are the only ones for which we could obtain data on the known vulnerabilities affecting them. Table 1 reports the main characteristics of the projects in our context—we report statistics on their change history with a particular focus on the refactoring operations observed.

2.3 Data Collection

This section describes how we collected each piece of information to address our research questions: developers' refactoring operations, security metrics, security-related technical debt, and known vulnerabilities that affected the projects considered.

Table 2: Set of refactoring types selected in this study collected using RefactoringMiner [24]. Each refactoring has a description and a comment on the expected impact on the source code security, which is also reported graphically in column 'H_p'.

Refactoring	Description	Expected Impact on Security	Hp
Package-Level			
Move Package	Moves a package between the source roots of the project.	Changing the package positioning should not affect any security-related aspect.	—
Class-Level			
Extract Superclass	Creates a shared superclass from a set of classes with common attributes and methods.	A superclass is commonly more accessible that subclasses, so it might expose previously hidden security-sensitive parts of the program, negatively affecting security.	↓
Extract Interface	Creates a shared interface from a set of classes with common methods.	An interface is commonly more accessible than its subclasses, but it should not change anything from implementing classes.	—
Move Class	Moves a class between the packages of the project.	Changing the belonging package should not affect any security-related aspect.	—
Method-Level			
Extract Method	Creates a new method containing part of the logic of an existing one, which will call the extracted method.	The new method of the extracted class might expose previously hidden security-sensitive parts of the program, negatively affecting security.	↓
Inline Method	Deletes a method and integrates its logic into all calling methods (i.e., the inverse of Extract Method).	The removal of a method may hide security-sensitive parts of the program, positively affecting security.	↑
Move Method	Moves a method between the classes of the project.	Changing the belonging class should not affect any security-related aspect.	—
Extract & Move Method	Successive application of Extract Method and Move Method refactorings.	Same as Extract Method.	↓
Move & Inline Method	Successive application of Move Method and Inline Method refactorings.	Same as Inline Method.	↑
Pull Up Method	Creates a shared method from a set of classes with common methods and places it in their superclass.	A new method in the superclass is commonly more accessible than its subclasses, so it might expose previously hidden security-sensitive parts of the program, negatively affecting security.	↓
Push Down Method	Removes a method of a superclass to place it in one of its subclasses.	The removal of a superclass method may hide security-sensitive parts of the program, positively affecting security.	↑
Attribute-Level			
Move Attribute	Moves an attribute between the classes of the project.	Changing the belonging class should not affect any security-related aspect.	—
Pull Up Attribute	Creates a shared attribute from a set of classes with common attributes and places it in their superclass.	A new attribute in the superclass is commonly accessed by subclasses through new accessor methods, negatively affecting security.	↓
Push Down Attribute	Removes an attribute of a superclass to place it in one of its subclasses.	The removal of a superclass attribute may remove accessor methods as well, positively affecting security.	↑

2.3.1 Mining Refactoring Data

We mined the entire change history of the considered projects to identify commits where developers applied at least one refactoring. To this aim, we run version 2.2 of RefactoringMiner [24] against each source code change. RefactoringMiner is a publicly available tool[5] that can detect a large number of refactoring types through the analysis of how the Abstract Syntax Tree of a Java class/method has changed with respect to the one of the previous commit. The output of RefactoringMiner is formatted as a JSON file reporting for each commit the set of refactoring operations applied and the classes/methods subject to them. Despite the existence of alternative refactoring detectors (e.g., RefDiff [45]), we opted for RefactoringMiner since it is publicly available and has a detection accuracy close to 100%, overcoming the capabilities of other detectors [24].

In the context of this study, we selected a set of common refactoring operations having mixed relations with security to uncover possible unexpected and sneaky correlations. Table 2 reports the 12 basic refactoring operations

[5]Link: `https://github.com/tsantalis/RefactoringMiner`

Table 3: List of security metrics computed by SURFACE.

Metric	Acronym	Description
Class-Level		
Classified Attributes	CA	Number of *classified* attributes of a class, identified through pattern matching heuristics (e.g. `password`, `token`).
Classified Methods	CM	Number of *classified* methods of a class, identified through (i) pattern matching heuristics (e.g.`validatePassword`, `generateToken`) or (ii) the check of usages of classified attributes.
Classified Instance Variables Accessibility	CIVA	Ratio of non-private and non-static classified attributes out of the total number of classified attributes (CA).
Classified Class Variables Accessibility	CCVA	Ratio of non-private and static classified attributes out of the total number of classified attributes (CA).
Classified Method Accessibility	CMA	Ratio of non-private classified methods out of the total number of classified methods (CM).
Classified Methods Ratio	CMR	Ratio of the number of classified methods (CM) out of all class methods.
Classified Attribute Interactions	CAI	Sum of the number of classified methods that access each classified attribute, divided by the product of the number of classified attributes and methods (CA × CM).
Project-Level		
Critical Classes	CC	Number of *critical classes*, i.e., classes with at least one classified components (classified attribute or method).
Critical Classes Ratio	CCR	Ratio of the number of critical classes (CC) out of all project classes.
Critical Classes Extensibility	CCE	Ratio of non-final critical classes out of the critical classes (CC).
Classified Methods Extensibility	CME	Ratio of non-final critical methods among all classes out of the critical methods among all classes.
Critical Super Classes Ratio	CSCR	Mean of the ratios, for each class, of the number of critical super classes out of all their super classes.
Serializable Critical Classes Ratio	SCCR	Ratio of serializable critical classes out of the critical classes (CC).

plus two composite refactoring operations (i.e., successive application of two elementary refactorings) we deemed worth investigating. Each row contains a description of how they work and of the possible impact on security. Such expected impacts derive from the refactoring operations' definition and represent the conjectures we aim to verify in our empirical investigation. In this respect, we formulated the hypotheses we posed for this study and described them graphically in column 'H_p'. An up arrow ('↑') indicates that we hypothesized a certain refactoring has a positive (good) effect on source code security; a down arrow ('↓') indicates that we hypothesized a negative (bad) effect. In contrast, the symbol '—' indicates a hypothesis of stability, i.e., the refactoring should not change the security profile of the source code in any way. As further explained in Section 2.5, these hypotheses were instantiated for the three specific research questions. All the selected refactoring operations alter the source code's internal structure at different granularity levels—ranging from individual attributes to groups of classes.

2.3.2 Mining Security Metrics

We computed a set of metrics that have been previously used to assess source code security [15, 46, 47] on all the refactoring commits of the projects. Table 3 reports their names and description. The metrics measure source code against the presence of confidential or sensitive information, e.g., user IDs, authorization tokens, or passwords, that might potentially worsen the security level. For instance, over-exposed (in terms of access specifiers) code fragments might lead to vulnerabilities that can be exploited.

To compute these metrics, we developed a tool, which we named SURFACE, and, for the sake of verifiability, we made it available in our online

appendix [28]. Our tool is a re-implementation of the one built by Abid et al. [15], as it was not publicly available. Given a source code file, it first verifies the presence of the security-related keywords identified in [15] to identify *"classified attributes"* (i.e., class fields that contain confidential or sensitive information), that will be used as a basis for applying further static analyses and computing the other metrics. To this end, we used a set of regular expressions based on those adopted by Abid et al. [15] to automatically detect all classified code elements (i.e., attributes, methods, and classes). The following box reports the regular expressions used by SURFACE as a comma-separated list of strings.

logins?, accounts?, auths?, authenticates?, authenticators?, auth[-_\s]?constraints?, roles?, permissions?, access(es)?, restricted, restricted[-_\s]?access(es)?, admins?, administrators?, certificates?, digital certificates?, fingerprints?, biometrics?, id, identifiers?, userid, uuid, client[-_\s]?ids?, user[-_\s]?ids?, username, user[-_\s]?details?, e[-_\s]?mail, passw(or)?ds?, pass[-_\s]?phrases?, pwds?, (secret[-_\s]?)?keys?, (api[-_\s]?)?tokens?, (oauth[-_\s]?)?tokens?, otp, credentials?, ip[-_\s]?address(es)?, ports?, hosts?, hostnames?, address(es)?, hiddens?, hidden[-_\s]?fields?, secrets?, top[-_\s]?secrets?, confidentials?, confidentiality, classified, privates?, private[-_\s]?fields?, private[-_\s]?members?, privacy, personals?, protect(ed)?, signatures?, (under)?cover(ed)?, payments?, credit[-_\s]?cards?([-_\s]?number)?s?, cards?, credits?, phone[-_\s]?numbers?, social[-_\s]?security[-_\s]?numbers?, date[-_\s]?of[-_\s]?birth, safe, (content[-_\s]?)?secure, security, security[-_\s]?management, security[-_\s]?constraint, sensitive, sensitive[-_\s]?data, sensitive[-_\s]?information, criticals?, vulnerables?, weaks?, weakness(es)?, backdoors?, (en)?crypt(ed)?, cipher([-_\s]?text)?, hash(ed)?, salt, nonce, encoded?, transcoded?, lock(ed)?, cach(ed)?, paths?, connection[-_\s]?string, transactions?, jobs?

We are aware that most of the metrics are derived from the set of classified attributes; hence they capture similar aspects connected to source code security. Yet, to the best of our knowledge, these are the only ones available that can enable an analysis of the security profile of object-oriented source code in a fully-automated fashion.

To collect the security metrics values, for each commit having at least one refactoring instance, we selected only those JAVA files directly involved in one of the refactorings made in the commit. Then, we run SURFACE twice: one time considering the files' versions before the commit, and one more time on the versions after the commit. The resulting metrics for the previous files' versions were subtracted from the latest versions, obtaining the *delta* (Δ) that represents how much a metric has changed in that commit. It is worth noting that for newly-added files the metrics were left as-is, while for the files deleted in the commit their values resulted to have a negative sign. Afterward, all the

Table 4: Top 10 most recurring SonarQube rules violated in the selected projects.

Security Rule	Description	#	Severity
Class Variable Visibility Check	Class variable fields should not have public accessibility	10,532	Critical
S1313	IP addresses should not be hardcoded	4,328	Critical
S1148	`Throwable.printStackTrace(...)` should not be called	4,193	Critical
S1444	Public static fields should be constant	3,732	Critical
S2386	Mutable fields should not be `public static`	1,306	Major
S2755	Fails for `DocumentBuilderFactory` XXE should be disabled	633	Blocker
S4423	Weak SSL/TLS protocols should not be used	484	Major
S2077	SQL binding mechanisms should be used	426	Major
S2068	Credentials should not be hard-coded	371	Critical
S5542	Encryption algorithms should be used with secure mode and padding scheme	333	Critical

class-level security metrics (Table 3) were aggregated to have individual values expressed at the entire commit level. Specifically, the deltas pertaining to the security metrics CA (Classified Attributes) and CM (Classified Methods) were summed together (as they represent counting), while the rest of the metrics were averaged. In this way, we could outline the magnitude of change in the security profile after a commit containing refactorings.

2.3.3 Mining Security-related Technical Debt

According to recent findings, SonarQube is among the most popular automated static analysis tools employed in practice [48, 49], in addition to being accurate when detecting security violations [50]. Based on these observations, we selected SonarQube version 7.5 to collect the metrics needed for $\mathbf{RQ}_2$. In particular, we measured the security-related technical debt collecting two kinds of metrics linked to the security profile of applications. On the one hand, for each refactoring commit, we counted the number of violations of security rules (i.e., those belonging to the *"Vulnerability"* group) that SonarQube encountered when analyzing the Java classes of the project's snapshot after the commit. Such violations indicate that the code is likely to be affected by a software vulnerability, or has laid the foundation for vulnerable code. Moreover, similarly to what we did for $\mathbf{RQ}_1$ (Section 2.3.2), we counted the violations only on those files directly involved in the refactorings that occurred in that commit. On the other hand, we obtained the so-called "security remediation effort", i.e., a measure of how much effort developers would spend when addressing all the detected violations in that snapshot—based on the violated security rules.

Once we collected all the technical debt-related metrics for all the refactoring commits, we computed the difference (Δ) between all the metrics values with their previous version (i.e., the parent commit) to compute the change in the number of violations and remediation effort—analogously to what we did for security metrics in $\mathbf{RQ}_1$. Overall, SonarQube was able to detect 26 different types of security violations. Table 4 reports the top 10 most recurring

Table 5: The 26 known vulnerabilities mined from NVD grouped by the 12 vulnerability types.

Vulnerability Type	Description	#CVE
CWE-264	Permissions, Privileges, and Access Controls	4
CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	4
CWE-287	Improper Authentication	3
CWE-79	Cross-site Scripting	3
NVD-CWE-noinfo	No sufficient information to classify the vulnerability	2
CWE-254	7PK Security Features	2
CWE-22	Path Traversal	2
CWE-352	Cross-Site Request Forgery (CSRF)	2
CWE-326	Inadequate Encryption Strength	1
CWE-310	Cryptographic Issues	1
CWE-20	Improper Input Validation	1
CWE-835	Infinite Loop	1

violations that were found and resolved within the refactoring commits we analyzed. Each rule in the table is accompanied by its description, the number of occurrences, and the corresponding severity level. The full list of rules for is available on the SonarQube website.[6]

2.3.4 Mining Known Vulnerability Data

We only considered vulnerabilities in NVD affecting the considered systems (see Table 1) and specifying the fixing commit (i.e., the one that officially patched a publicly disclosed vulnerability)—otherwise, we could not address our $\mathbf{RQ}_3$, as explained later in Section 2.4. From an operational perspective, we mined the full dump of NVD exploiting CVE-Search project [51], allowing the download of a JSON file containing all CVE records updated daily. We obtained the full JSON dump on May 30, 2022. We performed some additional filtering steps to remove incomplete/incorrect data that might have biased our observations: (1) we discarded CVEs that reported commits to different GitHub projects than those considered since we could not establish where the vulnerability was residing; (2) we filtered out vulnerabilities whose fixes were marked as `merge` commits, as these do not apply any real modification in the project history but simply incorporate the changes (i.e., a set of commits) from a branch into another, i.e., we could not consider them as actual patches since we were interested in getting precise information about the moment when fixes were added into the history rather than the moment when they were sent into the main branch. After this filtering, we ended up with a total of 26 known vulnerabilities of 12 distinct types, pertaining to nine NVD projects. Table 5 reports the 26 vulnerabilities grouped by their vulnerability type (CWE).

[6]Link: `https://rules.sonarsource.com`

2.4 Data Analysis

After collecting the data required to address our research questions, we proceeded with the statistical modeling and the subsequent interpretation.

2.4.1 $\boldsymbol{RQ_1}$-$\boldsymbol{RQ_2}$. Refactoring vs. Security-related Metrics and Technical Debt.

The first two research questions aim at understanding the effect of refactoring on indicators of longer-term source code security issues. For both **RQ**s, we employed similar analysis methods.

Starting from the security metrics (Section 2.3.2) and the security-related technical debt (Section 2.3.3) variations observed in the refactoring commits, we converted all Δ values into categories that could be better interpreted by humans. If a metric m had a $\Delta > 0$ in one of the refactoring commits analyzed, the variation was converted into the category *"Increased"*. Similarly, if m has a $\Delta < 0$, it was converted to the category *"Decreased"*. Otherwise, it was converted to *"Stable"*. It is worth noting that the interpretation of these categories depends on the specific metrics. Let us consider CA (Classified Attributes) metrics as an example. An increased number of classified attributes is generally deemed as something negative, as it indicates an increment in the number of fields holding security-sensitive data. In this case, observing many deltas labeled as *"Increased"* is a negative indication of the security profile of the application.

Afterwards, to address $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$ we built a statistical model for each security metric and technical debt in which we relate the number of distinct refactoring operations applied between c_{r-1} and c_r as well as other control variables to the three categories mentioned above, i.e., *"Increased"*, *"Stable"*, and *"Decreased"*. Approaching the research questions in this manner allowed us to verify which refactoring types have connections to security indicators and whether the effect of those refactoring operations is positive or negative.

More specifically, we considered the categorical values associated with each refactoring commit as *dependent variables*. The number of refactoring operations for each of the 14 considered types were treated as our *independent variables* in all the models. Furthermore, we computed three additional metrics that acted as the *confounding variables*, namely the factors that might significantly influence a dependent variable regardless of the values of the independent variables [52]. They are:

1. The number of lines of code (LOC) of the files' versions that underwent to refactoring, i.e., immediately before the commit detected by RefactoringMiner. All the LOC values were averaged to have a single summarized value for an entire commit. This metric has often been associated with a reduction of source code quality, and dependability [53, 54, 55]. The inclusion of this confounding factor was motivated by the assumption that working

on files with many lines of code might have a higher chance of increasing the values of security indicators or contributing to the introduction of vulnerabilities with respect to smaller files.

2. The Weighted Methods per Class (WMC) [56] computed on the files' versions that underwent refactoring, i.e., immediately before the commit detected by RefactoringMiner. All the WMC values were averaged to have a single summarized value for an entire commit. This metric represents the sum of McCabe's cyclomatic complexity [57] values computed on the class's methods. In this case, the negative impact of code complexity on vulnerabilities has been previously assessed [58, 59].
3. The code churn, i.e., the amount of code added/deleted in the commit that touched the files' that underwent to refactoring. All the churn values were summed to have a single summarized value for an entire commit. Previous work has shown that the higher the churn of two subsequent commits, the higher the likelihood to introduce issues in the code [60]. The negative impact of churn metrics has also been assessed when considering software vulnerabilities [61].

Such metrics were extracted using PyDriller [62], which allows straightforward analyses of projects' change history, and Lizard,[7] which parses the source code and automatically extracts a set of common structural metrics from the source code.

Furthermore, we encoded the projects as 39 different binary variables to capture any possible random effect coming from a specific project.

Having a categorical dependent variable, we fit a mixed-effect Multinomial Log-Linear model [33], a classification method that can generalize logistic regression to multiclass problems, so fitting our case. The models were built using the R toolkit exploiting the `multinom` model of the package `nnet`.[8]

The choice of a mixed-effect Multinomial Log-Linear model was driven by multiple observations. First and foremost, it fits the multiclass problem we intended to model when building a theory of how refactoring is related to security. Second, it outputs precious pieces of information that can be used to interpret the results, as detailed in the remainder of the section. It indeed provides statistical codes through which each individual refactoring type can be assessed against its statistical relevance for the problem under analysis—as such, we could identify the refactoring types having a statistically significant connection to security. Furthermore, it returns the *odds ratios* (OR) [63]—i.e., the exponential of the model's coefficients—that provide a measure of the actual impact of the associated variables, i.e., the refactoring type. Such interpretation complements the statistical codes, providing a more practical measure to interpret the effects of refactoring on security. Other research methods, e.g., correlation analysis, cannot provide such a comprehensive and tangible assessment of our hypotheses. Perhaps more importantly, it is important to remark

[7] `https://pypi.org/project/lizard`

[8] `https://cran.r-project.org/web/packages/nnet/nnet.pdf`

that security might and might not be affected by the refactoring; other factors might play a role. The statistical modeling exercise allowed us to specify a set of confounding variables and, for this reason, assess the impact of refactoring while keeping other factors into account.

When building the models, we took the problem of multicollinearity into account. This arises in cases where two or more independent variables are linearly correlated, and one can be predicted from the other, possibly biasing the model's fitting capabilities and how the results are interpreted. In this respect, we first verified the normality of the distributions of the independent variables employing the Anderson-Darling normality test [64]. Such a test verifies whether a given sample follows a theoretical distribution, i.e., the normal one. For each independent variable, we compared its distribution with a normal distribution having the same mean and standard deviation of the sample. As a result, all the test runs failed to reject the null hypothesis, hence indicating that our data are not normally distributed. Because of the non-normality of any of the independent variables samples, we computed the Spearman's rank correlation [65] between all possible pairs of independent variables to determine whether there are strongly correlated pairs (i.e., variables for which the Spearman's $\rho > 0.8$). This step did not eventually find any correlated variables, meaning that the independent variables' distribution was different enough to be used together in the statistical models.

As for the interpretation of the results, it is worth noting that the model's logit coefficients c_i are relative to a reference category and indicate how the independent variables vary the chances of the dependent variable being affected with respect to the reference category. We set such a category to *"Stable"* to estimate how the various independent variables, i.e., the refactoring operations, likely change in *either* a positive or negative direction the stability of security indicators. For instance, if we have the refactoring type r_i that presents a logit coefficient $c_i = -1.50$ in the model built when analyzing the decrease in the security metric s_j, this means that a one-unit increase of r_i would lead to an increase of the chances of s_j to remain stable.

After obtaining the logit coefficients, we computed the odds ratios (ORs) using the exponential function (e^{c_i}). In our case, the ORs complement the interpretation of the results: for an independent variable, i.e., a refactoring type, it indicates the increment of chances for a class to increase/decrease the value of a security metric ($\mathbf{RQ}_1$) or a technical debt metric ($\mathbf{RQ}_2$) as a consequence of a one-unit increase of the refactoring. With the OR values, we could quantify the extent to which the application of refactoring impacts security metrics and debt, hence giving a more practical sense to the coefficients obtained when running the models.

2.4.2 $\boldsymbol{RQ}_3$. Refactoring vs. Known Vulnerabilities

The last research question measures the extent to which refactoring operations contribute to the introduction of vulnerabilities. To address it, the first chal-

lenge was concerned with mining the commits responsible for the introduction of vulnerabilities.

To obtain these *vulnerability-contributing commits* (a.k.a. VCCs) [66], we have followed the idea behind the well-known SZZ algorithm [34], which recovers the set of commits that likely introduced a defect starting from a bug-fixing commit using the `git-blame` functionality on the lines deleted during the fix. Despite SZZ has been envisioned to fetch the commits that induce traditional defects [67], it has also been exploited to fetch vulnerability-contributing commits [68, 69, 70]. Yet, it has been subject to adjustments and improvements. In this work, we adopted a set of heuristics to reduce the amount of noisy results. Specifically, for each JAVA file F_i modified in a vulnerability-fixing commit f, we run the `git-diff` functionality to obtain the list of added and deleted lines in F_i and then applied two strategies to obtain the VCCs. Firstly, we run the `git-blame` command to obtain the commits that last changed the lines deleted in f. Secondly, we blamed the lines "around" continuous blocks of changes—generally representing new checks—made only of added lines. The former was done to recover those commits that have likely added flawed pieces of code—e.g., a call to an improper input validation function or the use of an obsolete cryptography algorithm. Instead, the latter can reach the commits touching the code areas that lacked solid control mechanisms. The only exception was made for blocks made of totally new functions or methods, as they can be placed anywhere, rendering their contextual lines irrelevant. In addition, we did not blame empty and comment lines, and irrelevant non-source code files—e.g., documentation, build, blob, and test files—as they do not generally contribute to a vulnerability. What is more, we did not consider the VCCs that merged changes from multiple commits, as they do not report real modifications per se. It is worth noting that vulnerabilities could have been fixed by multiple fixing commits; in such cases, we united the set of VCCS obtained from each fixing commit to build the final set. The described procedure was implemented exploiting PYDRILLER [62] repository mining library with the help of the parsing library LIZARD[9] to apply our heuristics.

Once we had detected the vulnerability-contributing commits, we could verify in how many cases such commits were also marked as refactoring commits (collected as described in Section 2.3.1). Therefore, we sought to elicit the amount of vulnerability-contributing commits for which refactoring might have played a role. When addressing $\mathbf{RQ}_3$, we also reported the results by considering each refactoring type individually, hence assessing if a particular operation is more likely to contribute to the introduction of a vulnerability.

2.5 Hypotheses and Statistical Verification

Once we completed the statistical modeling, we proceeded with the verification of the high-level hypotheses formulated in Table 2. More specifically, we

[9] `https://pypi.org/project/lizard`

first refined them to derive more concrete null and alternative hypotheses to test the research questions in this study. In the cases of *Move Package*, *Extract Interface*, *Move Class*, *Move Method*, and *Move Attribute* refactoring, we defined the following null hypothesis:

Hn_1 The refactoring has a *significant impact*, either positive or negative, on security properties.

Our alternative hypothesis was, instead:

Ha_1 There is *no significant impact* of the refactoring on security properties.

In the context of $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$, we rejected the null hypotheses if the coefficients of the statistical models built to understand the *increase* and *decrease* of the security properties were not significant or negative. In the latter case, the coefficients of the Multinomial Log-Linear model would indicate that refactoring operations tend to increase the likelihood of security metrics/debt being stable, hence rejecting the null hypothesis in favor of the alternative one. As for $\mathbf{RQ}_3$, we run the non-parametric Mann-Whitney U test [71] (with $\alpha = 0.5$) on the distribution of refactoring operations within VCCs and non-VCCs commits. We rejected the null hypothesis if $\alpha > 0.05$. We also measured the effect size of the differences identified in the two distributions using Cohen's d [72]. We followed well-established thresholds for interpretation: 0.2 for *Small*, 0.5 for *Medium* and 0.8 for *Large* effect size [72].

With respect to *Extract Superclass*, *Extract Method*, *Extract & Move Method*, *Pull Up Method*, and *Pull Up Attribute* refactoring, our null hypothesis was:

Hn_2 There is *no significant impact* of the refactoring on security properties.

The alternative hypothesis in this case was:

Ha_2 The refactoring has a *significant negative impact* on security properties.

In $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$, we rejected the null hypothesis if we observed (i) both significant coefficients and (ii) positive coefficients in the statistical model built to understand the *increase* of security metric/debt values and negative coefficients in the statistical model built to understand the *decrease* of security metric/debt values. In the latter case, indeed, the statistical model coefficients would tell us that the application of refactoring operations tends to increase the likelihood of the metrics/debt being increased, hence indicating their deterioration. As for $\mathbf{RQ}_3$, we still relied on the same outcomes and interpretation of the Mann-Whitney U test [71] and Cohen's d [72].

Finally, when it comes to *Inline Method*, *Move & Inline Method*, *Push Down Method*, and *Push Down Attribute* refactoring, the null hypothesis was set to:

Hn_3 There is *no significant impact* of the refactoring on security properties.

The alternative hypothesis in this case was:

Ha_3 The refactoring has a *significant positive impact* on security properties.

As for $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$, we rejected the null hypothesis if we observed (i) both significant coefficients and (ii) positive coefficients in the statistical model built to understand the *decrease* of security metric/debt values and negative coefficients in the statistical model built to understand the *increase* of security metric/debt values. In the latter case, the statistical coefficients would indicate that the application of refactoring operations has the tendency to increase the likelihood of the metrics/debt being decreased, which would mean that the security profile would improve. In $\mathbf{RQ}_3$, we relied on the outcomes and interpretation of the Mann-Whitney U test [71] and Cohen's d [72].

2.6 Verifiability and Replicability

In order to allow our study to be verified and replicated, we have published the complete raw data, along with the data collection and analysis scripts in our online appendix [28]. The `'README.md'` file contains more precise instructions on how to use our artifacts to replicate the study.

3 Analysis of the Results

In this section, we report the results of the empirical study, discussing them by research question.

3.1 $\mathbf{RQ}_1$. To what extent do refactoring operations impact security metrics?

In $\mathbf{RQ}_1$, we sought to understand the relation between refactoring and security metrics. Table 6 reports for each refactoring type the sign of the logit coefficients (within a circle) and the value of the ORs obtained for the Multinomial Log-Linear models built to understand the decrease and increase of the security metrics considered in the study. The coefficients of the variables that turned out to be statistically significant are reported with a colored symbol (green for ⊕, red for ⊖), otherwise are left white. In addition, the cells with a gray shade indicate that the impact of that type of refactoring rejects the null hypothesis formulated in Section 2.5—i.e., the impact turned out to be in line with our expectations. For the sake of readability, we did not report the coefficient signs and ORs values of the confounding factors (LOC, WMC, and code churn) considered when building the models, but we discuss their role in our analysis and report the full results in our online appendix [28]. Looking at the table, we can immediately observe that the refactoring types *Move Method*, *Move Attribute*, *Extract Superclass*, and, to a lesser extent, *Push Down Method*, always had positive coefficients in all the 13 models built for the 13 security metrics; this means that moving code components (i.e., attributes or methods) from a class to another or optimizing the degree of code reuse

Table 6: The impact of each refactoring type on security-related metrics ($\mathbf{RQ}_1$) represented via the sign of the models' coefficients (colored if $p < 0.05$) and their odds ratios. The category 'DECR.' represents the cases where $\Delta < 0$, while 'INCR.' represents $\Delta > 0$. The cells in gray indicate the acceptance of the related alternative hypotheses (Ha_1) formulated for $\mathbf{RQ}_1$. (Section 2.5).

Metric	Categories		Categories		Categories		Categories	
	DECR.	INCR.	DECR.	INCR.	DECR.	INCR.	DECR.	INCR.
	Extract Method		**Inline Method**		**Extract & Move Met.**		**Move & Inline Met.**	
CA	− 0.882	+ 1.021	+ 1.114	− 0.954	− 0.991	− 0.989	− 0.997	+ 1.005
CAI	+ 1.021	+ 1.013	+ 1.061	+ 1.035	+ 1.002	− 0.996	+ 1.001	+ 1.005
CC	− 0.940	+ 1.024	+ 1.227	+ 1.016	− 0.989	− 0.988	+ 1.002	− 0.993
CCE	− 0.971	+ 1.013	+ 1.271	+ 1.063	+ 1.016	+ 1.002	+ 1.003	− 0.993
CCR	− 0.805	− 0.930	+ 1.010	− 0.959	− 0.938	− 0.986	+ 1.006	− 0.997
CCVA	+ 1.009	+ 1.025	+ 1.030	+ 1.141	− 0.993	− 0.996	+ 1.003	− 0.994
CIVA	− 0.999	− 0.947	− 0.975	− 0.995	+ 1.002	− 0.971	− 0.995	− 0.993
CM	− 0.861	+ 1.043	+ 1.140	− 0.862	− 0.992	− 0.988	− 0.997	+ 1.004
CMA	+ 1.035	− 1.000	+ 1.041	+ 1.045	− 0.997	− 0.994	− 0.997	+ 1.004
CME	− 0.935	+ 1.057	+ 1.122	− 0.993	− 0.996	− 0.988	− 0.999	− 0.999
CMR	+ 1.013	+ 1.015	+ 1.031	+ 1.063	+ 1.019	− 0.999	+ 1.087	+ 1.089
CSCR	− 0.839	− 0.928	− 0.982	+ 1.041	− 0.995	+ 1.005	+ 1.006	+ 1.003
SCCR	+ 1.029	− 0.951	+ 1.040	+ 1.196	− 0.982	− 0.998	− 0.992	+ 1.000
	Move Package		**Move Class**		**Move Method**		**Move Attribute**	
CA	+ 1.353	+ 1.184	+ 1.002	+ 1.004	+ 1.063	+ 1.032	+ 1.148	+ 1.171
CAI	+ 1.486	+ 1.416	− 0.996	+ 1.005	+ 1.048	+ 1.043	+ 1.230	+ 1.274
CC	+ 1.536	+ 1.249	− 0.994	+ 1.004	+ 1.076	+ 1.049	+ 1.222	+ 1.222
CCE	− 0.944	+ 1.245	− 0.998	+ 1.004	+ 1.059	+ 1.043	+ 1.210	+ 1.177
CCR	+ 1.022	+ 1.221	− 0.987	+ 1.002	+ 1.180	+ 1.184	+ 1.460	+ 1.489
CCVA	+ 1.772	+ 1.228	− 0.956	+ 1.013	+ 1.016	+ 1.017	+ 1.141	+ 1.155
CIVA	− 0.941	− 0.836	+ 1.002	+ 1.009	+ 1.044	+ 1.042	+ 1.080	+ 1.109
CM	+ 1.327	+ 1.210	− 0.996	+ 1.001	+ 1.055	+ 1.016	+ 1.129	+ 1.121
CMA	+ 1.356	+ 1.188	− 0.999	+ 1.001	+ 1.043	+ 1.048	+ 1.123	+ 1.166
CME	+ 1.827	− 0.788	− 0.919	+ 1.003	+ 1.061	− 0.992	+ 1.095	+ 1.102
CMR	+ 1.253	+ 1.107	− 0.995	+ 1.002	+ 1.031	+ 1.034	+ 1.097	+ 1.136
CSCR	− 0.629	+ 1.656	+ 1.009	+ 1.006	+ 1.168	+ 1.148	+ 1.463	+ 1.459
SCCR	+ 1.447	+ 1.496	+ 1.005	+ 1.005	+ 1.048	+ 1.059	+ 1.232	+ 1.231
	Pull Up Method		**Pull Up Attribute**		**Push Down Method**		**Push Down Attribute**	
CA	− 0.996	+ 0.952	+ 1.044	+ 1.006	+ 1.071	+ 1.043	+ 0.865	− 0.960
CAI	+ 1.011	− 0.994	− 0.984	− 0.995	+ 1.053	+ 1.058	− 0.984	+ 1.002
CC	− 0.994	− 0.973	+ 1.066	+ 1.037	+ 1.052	+ 1.018	+ 1.007	− 0.999
CCE	− 0.994	− 0.980	+ 1.065	+ 1.038	+ 1.040	+ 1.031	+ 1.018	+ 1.004
CCR	+ 1.013	+ 1.003	− 0.996	− 0.998	+ 1.039	+ 1.042	+ 1.009	+ 1.009
CCVA	− 0.984	− 0.985	+ 1.048	+ 1.046	+ 1.045	+ 1.072	− 0.941	− 0.880
CIVA	+ 1.021	− 0.998	− 0.986	+ 1.010	+ 1.073	+ 1.065	− 0.681	− 0.958
CM	+ 1.006	− 0.972	+ 1.017	+ 1.013	+ 1.052	+ 1.026	− 0.917	− 0.982
CMA	+ 1.007	− 0.990	+ 1.005	− 0.999	+ 1.057	+ 1.043	− 0.849	− 0.968
CME	+ 1.017	− 0.992	− 0.986	− 0.997	+ 1.037	− 0.987	− 0.999	+ 1.011
CMR	+ 1.002	− 0.998	+ 1.004	− 0.999	+ 1.021	+ 1.025	− 0.952	− 0.993
CSCR	+ 1.079	+ 1.072	− 0.877	− 0.997	+ 1.070	+ 1.061	+ 1.020	− 0.997
SCCR	− 0.996	− 0.967	+ 1.024	+ 1.088	+ 1.038	+ 1.054	− 1.000	+ 1.005
	Extract Superclass		**Extract Interface**					
CA	+ 2.071	+ 2.629	+ 1.054	+ 1.151				
CAI	+ 1.965	+ 2.378	+ 1.206	+ 1.072				
CC	+ 1.632	+ 2.295	− 0.711	+ 1.008				
CCE	+ 1.487	+ 2.254	− 0.770	+ 1.042				
CCR	+ 17.743	+ 8.504	− 0.915	+ 1.046				
CCVA	+ 1.846	+ 1.550	+ 1.074	− 0.742				
CIVA	+ 1.722	+ 2.744	+ 1.201	− 0.637				
CM	+ 1.764	+ 1.678	+ 1.046	− 0.973				
CMA	+ 1.460	+ 1.950	+ 1.056	+ 1.185				
CME	+ 1.297	+ 1.178	+ 1.076	+ 1.200				
CMR	+ 1.190	+ 1.666	− 0.879	+ 1.030				
CSCR	+ 13.387	+ 13.125	+ 3.263	+ 1.382				
SCCR	+ 2.026	+ 2.120	− 0.996	− 0.646				

(i.e., creating better class hierarchies) have the effect of varying the security profile of an application, either positively or negatively. Moreover, the *Extract Superclass* refactoring had a particularly strong impact on CCR and CSCR metric, i.e., their ORs are the largest among all the other models. As such, extracting new classes in hierarchies impacts (i) the ratio of the number of crit-

ical classes to the number of classes in the entire project (ii) and the number of superclasses in all the class hierarchies (CSCR), respectively. Although these findings suggest that these refactoring types have a random effect on security metrics, it is reasonable to believe that the impact is determined by their specific application, i.e., the security profile is affected differently depending on how developers apply the refactoring operations.

Surprisingly, refactoring types for which we expected no impact, such as *Extract Interface* and *Move Package*, still exhibited a statistically significant mixed effect on the security metrics. Between the two, *Extract Interface* showed a clearer behavior. On the one hand, it tends to increase the value of CC, CCE, CCR, and CMR metrics—i.e., increasing the number of critical classes. On the other hand, it keeps reducing CCVA, CIVA, and CM metrics—i.e., reducing both the accessibility of instance and class variables, and the number of classified methods. This means that re-organizing the classes' interfaces helps keep the number of critical attributes and methods under control, still increasing the risk of introducing too many critical classes. Differently, only for the CIVA metric, the *Move Package* refactoring matched our expectations: moving classes among packages does not affect this metric at all. This might be explained by the fact that commits applying package restructuring are generally not done in a fully-isolated manner but are applied in the context of other changes—which have a mixed impact.

Extract Method, *Inline Method*, and *Move Class* still exhibited mixed effects on the various security metrics, but with much lesser significance than other refactoring types. The only cases where the null hypotheses were rejected were for *Extract Method* for CM and CME metrics, and also for *Inline Method* for CM metric. This is quite straightforward to comprehend. *Extract Method* creates new methods from a piece of code in existing methods, likely introducing new classified methods if the extracted logic deals with classified attributes. At the same time, *Inline Method* refactoring eliminates redundant methods, with a high chance of removing classified methods and reducing access to classified attributes. This was the case of project Conversation at the revision `14cfb609`. In such a commit, the utility class `CryptoHelper` was streamlined into three new classes, all placed in package `crypto/sasl`. Additionally, the `XmppConnection` class—in change or managing XMPP connections—was refactored to inline the two methods `sendSaslAuthPlain()` and `sendSaslAuthDigestMd5()` into `processStreamFeatures()`. Indeed, the two removed methods were only called by `processStreamFeatures()`, so driving the developer to apply two *Inline Method* refactorings—the commit message confirms this intentions, i.e., *'Refactor authentication code'*. In this respect, `sendSaslAuthPlain()` was a classified method, as it access to security-sensitive data, i.e., `account` instance variable in this case. Hence, its removal led to the reduction of one from the CM metric. This variation translates into a reduction of the overall application's attack surface. Indeed, the metrics proposed by Alshammari et al. [73] penalize those classes exposing too many methods that access classified components, as attackers might leverage them to carry out attacks. This explains why *Inline Method* refactorings have

been seen as beneficial from this perspective is beneficial. Such an example also opens an interesting observation. While a securely-designed class should minimize the number of methods with the responsibility of accessing security-critical data, good design practices for maintainable code recommend creating many small and cohesive methods. Thus, creating both maintainable and secure code demands particular care not to create too many classified methods but still avoiding making poorly cohesive and long methods. In other words, accessing security-critical data should be reserved for only an essential set of elected classes and methods. Similarly, in commit `43531113` of the same project, an *Extract Method* refactoring was applied on `sendBindRequest()` to extract `sendIqPacket()` method, which happened to access security-sensitive data—so, it was branded as a new classified method. While this method is not necessarily a security issue, its presence increases the application's attack surface, giving attackers an additional method to leverage for its purpose.

All the other refactoring types, i.e., *Move Class*, *Pull Up Attribute*, *Push Down Attribute*, and *Pull Up Method*, appears to have limited or no impact on any security metrics. Perhaps more interestingly, the composite refactoring types considered in the study, namely *Extract & Move Method* and *Move & Inline Method*, seem to show a "mixture" of the behaviors of their individual operations. This had the curious effect of having almost no statistical significance for all 13 models. Yet, this could also be caused by the limited amount of composite refactorings observed in our dataset.

When considering the impact of the confounding factors, i.e., LOC, WMC, and code churn, we could notice that only code churn has a statistically significant relationship with all dependent variables. On the contrary, WMC turned out to have a poor impact on the security metrics, indicating that the complexity of methods is not related to the number of critical components in the source code. All in all, the ORs for all confounding factors are still very low, translating into a very weak effect on security.

Last but not least, we also found that in some cases, the projects themselves turned out to be significant for the explanation of the dependent variable. From a practical point of view, this means that the peculiarities of the projects have some influence on the changes to the global security profile. Our study cannot uncover the reasons behind this finding, as it would deserve further investigation. Yet, it is reasonable to believe that project-specific properties exist, e.g., contribution guidelines [74], code of conducts [38], and more, that make developers more or less prone to introduce vulnerabilities.

Main findings for $\mathbf{RQ}_1$

Different refactoring types have a different impacts on security metrics. In most cases, there are just variations, in either positive or negative ways, without a clear direction. Refactoring types such as *Inline Method* or *Extract Interface* may help keep the number of classified attributes and methods under control. The application of refactoring sequences causes a mixture of the effect of the individual components. More in general, the effect on security seems to depend on how refactoring operations are applied.

3.2 $\mathbf{RQ}_2$. To what extent do refactoring operations impact security-related technical debt?

In $\mathbf{RQ}_2$, we investigated the relation between refactoring and security-related technical debt, measured through the number of security violations detected by SonarQube and the security remediation effort. Similarly to $\mathbf{RQ}_1$, Table 7 reports the sign of the logit coefficients (within a circle) and the value of the ORs obtained for the Multinomial Log-Linear models for each refactoring type. Here too, the coefficients of the variables that turned out to be statistically significant are reported with a green ⊕ or red ⊖, otherwise are left white. Whenever the impact of a refactoring type rejects a null hypothesis formulated in Section 2.5 the related cells are depicted in gray. The table does not report the confounding factors (LOC, WCM, and code churn), but these are reported in the raw results in our online appendix [28].

Looking at the results, we could immediately notice the predominant presence of negative coefficients associated with both the decrease or increase of rule violations, implying that the majority of refactoring operations tend to keep the number of violations stable. This is particularly evident for *Extract Interface*, *Move Package*, *Push Down Method*, and *Push Down Attribute*. In other words, these refactoring types generally do not introduce or resolve any security-related violation. This is in line with their definition. *Extract Interface* and *Move Package* do not overhaul the code structure of the involved classes, so it is reasonable that they do not affect any security rule. The refactoring operations that push attributes or methods down to class hierarchies—i.e., *Push Down Method*, *Push Down Attribute*—do not seem to affect security rules in any way. However, both *Move Package* and *Move Attribute* are connected to a variation of the security remediation effort. This could be explained by the fact that these kinds of changes are made in conjunction with other changes that introduce security-related technical debt.

Despite these results, there are some notable exceptions worth analyzing. *Extract Superclass* refactoring significantly increases the chance of increasing the security remediation effort. Such a refactoring type also tends to violate rule S2386, i.e., *'Mutable fields should not be* `public static`*'*. Violating such a rule has a security implication, as the presence of `public static` fields expose mutable objects or arrays to changes by malicious users. Such bad

Table 7: The impact of each refactoring type on security debt and violations ($\mathbf{RQ}_2$) represented via the sign of the models' coefficients (colored if $p < 0.05$) and their odds ratios. The category 'DECR.' represents the cases where $\Delta < 0$, while 'INCR.' represents $\Delta > 0$. The cells in gray indicate the acceptance of the related alternative hypotheses (Ha_2) formulated for $\mathbf{RQ}_2$. (Section 2.5).

Metric	Categories		Categories		Categories		Categories	
	DECR.	INCR.	DECR.	INCR.	DECR.	INCR.	DECR.	INCR.
	Extract Method		**Inline Method**		**Extract & Move Met.**		**Move & Inline Met.**	
SRE	⊕ 1.016	⊕ 1.028	⊕ 1.044	⊕ 1.038	⊖ 0.982	⊖ 0.993	⊕ 1.001	⊕ 1.004
S1989	⊖ 0.000	⊖ 0.751	⊕ 1.418	⊖ 0.000	⊖ 0.000	⊕ 1.475	⊖ 0.967	⊖ 0.000
S2647	⊕ 1.145	⊖ 0.480	⊖ 0.022	⊖ 0.003	⊖ 0.021	⊕ 1.208	⊖ 0.012	⊖ 0.010
S2755	⊕ 1.100	⊖ 0.874	⊕ 1.180	⊖ 0.560	⊖ 0.953	⊖ 0.828	⊖ 0.000	⊕ 1.027
S2976	⊖ 0.825	⊖ 0.891	⊕ 1.269	⊕ 1.394	⊖ 0.900	⊕ 1.123	⊕ 1.150	⊖ 0.000
S4423	⊖ 0.829	⊖ 0.911	⊖ 0.000	⊕ 1.331	⊖ 0.250	⊕ 1.061	⊖ 0.842	⊖ 0.000
S4830	⊖ 0.896	⊖ 0.718	⊖ 0.000	⊖ 0.000	⊖ 0.000	⊖ 0.851	⊖ 0.000	⊖ 0.000
S5527	⊖ 0.959	⊖ 0.557	⊖ 0.000	⊖ 0.452	⊖ 0.000	⊖ 0.633	⊖ 0.000	⊖ 0.002
S5542	⊖ 0.572	⊕ 1.059	⊕ 1.205	⊕ 1.196	⊕ 1.253	⊕ 1.037	⊖ 0.000	⊕ 1.012
S5547	⊖ 0.000	⊖ 0.221	⊖ 0.000	⊖ 0.000	⊕ ∼965	⊖ 0.000	⊖ 0.000	⊖ 0.339
S2068	⊖ 0.0933	⊖ 0.886	⊖ 0.000	⊖ 0.000	⊖ 0.000	⊖ 0.000	⊖ 0.000	⊖ 0.000
S2386	⊖ 0.019	⊖ 0.993	⊖ 0.027	⊖ 0.962	⊖ 0.047	⊕ 1.004	⊖ 0.138	⊖ 0.995
	Move Package		**Move Class**		**Move Method**		**Move Attribute**	
SRE	⊕ 1.845	⊕ 1.476	⊖ 0.983	⊕ 1.004	⊕ 1.007	⊕ 1.004	⊕ 1.068	⊕ 1.053
S1989	⊖ 0.030	⊖ 0.001	⊖ 0.002	⊕ 1.313	⊖ 0.000	⊖ 0.000	⊖ 0.000	⊖ 0.000
S2647	⊖ 0.412	⊖ 0.007	⊖ 0.238	⊕ 1.016	⊖ 0.032	⊖ 0.516	⊕ 1.392	⊖ 0.005
S2755	⊖ 0.000	⊖ 0.000	⊖ 0.826	⊖ 0.727	⊖ 0.901	⊕ 1.018	⊕ 1.502	⊕ 1.136
S2976	⊖ 0.001	⊖ 0.000	⊖ 0.985	⊖ 0.475	⊖ 0.954	⊕ 1.136	⊖ 0.618	⊕ 1.150
S4423	⊖ 0.010	⊕ 9.633	⊖ 0.600	⊖ 0.939	⊖ 0.809	⊕ 1.013	⊖ 0.955	⊖ 0.915
S4830	⊖ 0.003	⊖ 0.015	⊖ 0.000	⊖ 0.902	⊖ 0.000	⊖ 0.662	⊖ 0.971	⊖ 0.000
S5527	⊖ 0.000	⊕ 0.000	⊖ 0.000	⊖ 0.913	⊖ 0.389	⊕ 1.038	⊕ 1.925	⊖ 0.765
S5542	⊖ 0.001	⊖ 0.100	⊖ 0.731	⊖ 0.626	⊕ 1.169	⊖ 0.985	⊖ 0.000	⊖ 0.489
S5547	⊖ 0.000	⊖ 0.000	⊖ 0.000	⊖ 0.717	⊖ 0.000	⊖ 0.000	⊖ 0.000	⊖ 0.000
S2068	⊖ 0.028	⊖ 0.001	⊖ 0.000	⊖ 0.055	⊖ 0.000	⊕ 1.032	⊖ 0.000	⊖ 0.861
S2386	⊕ 2.985	⊕ 1.323	⊖ 0.518	⊖ 0.972	⊕ 1.942	⊖ 0.986	⊖ 0.046	⊕ 1.102
	Pull Up Method		**Pull Up Attribute**		**Push Down Method**		**Push Down Attribute**	
SRE	⊖ 0.999	⊖ 0.999	⊕ 1.022	⊕ 1.001	⊖ 0.973	⊖ 0.996	⊕ 1.023	⊕ 1.041
S1989	⊖ 0.000	⊖ 0.696	⊖ 0.000	⊕ 4.714	⊖ 0.000	⊖ 0.003	⊖ 0.011	⊖ 0.031
S2647	⊖ 0.282	⊖ 0.009	⊖ 0.434	⊖ 0.147	⊖ 0.336	⊖ 0.098	⊖ 0.842	⊖ 0.536
S2755	⊕ 1.265	⊖ 0.997	⊖ 0.359	⊖ 0.956	⊖ 0.000	⊕ 1.671	⊖ 0.000	⊖ 0.000
S2976	⊖ 0.769	⊖ 0.939	⊖ 0.000	⊕ 1.280	⊖ 0.000	⊖ 0.000	⊖ 0.000	⊖ 0.000
S4423	⊖ 0.825	⊖ 0.904	⊕ 1.366	⊖ 0.000	⊕ 1.088	⊖ 0.000	⊕ 1.002	⊖ 0.000
S4830	⊖ 0.000	⊖ 0.000	⊖ 0.000	⊖ 0.000	⊖ 0.000	⊖ 0.000	⊖ 0.000	⊖ 0.000
S5527	⊖ 0.877	⊖ 0.000	⊕ 1.910	⊖ 0.000	⊖ 0.000	⊖ 0.000	⊖ 0.000	⊖ 0.000
S5542	⊖ 0.000	⊖ 0.719	⊖ 0.000	⊕ 1.155	⊖ 0.000	⊕ 1.550	⊖ 0.000	⊖ 0.000
S5547	⊖ 0.000	⊖ 0.013	⊖ 0.000	⊖ 0.000	⊖ 0.000	⊖ 0.000	⊖ 0.000	⊖ 0.000
S2068	⊖ 0.000	⊖ 0.682	⊖ 0.000	⊕ 1.411	⊖ 0.000	⊖ 0.030	⊖ 0.152	⊖ 0.063
S2386	⊖ 0.200	⊖ 0.944	⊖ 0.340	⊕ 1.005	⊖ 0.025	⊕ 1.054	⊖ 0.206	⊕ 1.107
	Extract Superclass		**Extract Interface**					
SRE	⊖ 0.901	⊕ 1.264	⊖ 0.994	⊕ 1.057				
S1989	⊖ 0.000	⊕ 7.920	⊖ 0.000	⊖ 0.000				
S2647	⊖ 0.005	⊖ 0.011	⊖ 0.119	⊖ 0.014				
S2755	⊕ 1.352	⊖ 0.944	⊖ 0.000	⊖ 0.353				
S2976	⊕ 1.521	⊖ 0.791	⊖ 0.000	⊖ 0.000				
S4423	⊕ 1.148	⊖ 0.708	⊕ 2.094	⊖ 0.001				
S4830	⊖ 0.000	⊖ 0.000	⊖ 0.001	⊖ 0.000				
S5527	⊖ 0.854	⊖ 0.000	⊖ 0.000	⊖ 0.000				
S5542	⊖ 0.000	⊕ 1.094	⊕ 3.360	⊕ 1.529				
S5547	⊖ 0.000	⊖ 0.000	⊖ 0.000	⊖ 0.000				
S2068	⊖ 0.000	⊖ 0.003	⊖ 0.000	⊖ 0.000				
S2386	⊖ 0.000	⊕ 1.585	⊖ 0.001	⊖ 0.874				

SRE stands for "Security Remediation Effort", measuring the security technical debt

practices are also categorized by CWE-582: *'Array Declared Public, Final, and Static'*, CWE-607: *'Public Static Final Field References Mutable Object'*, and CWE-766: *'Critical Data Element Declared Public'*, all representing weaknesses in the source code that attackers can leverage to carry out attacks. In other words, mutable objects should not be leaked to client programs as they can violate class invariants and disrupt the normal execution flow of the target application—if they have access to its runtime. Moreover, violations of this rule

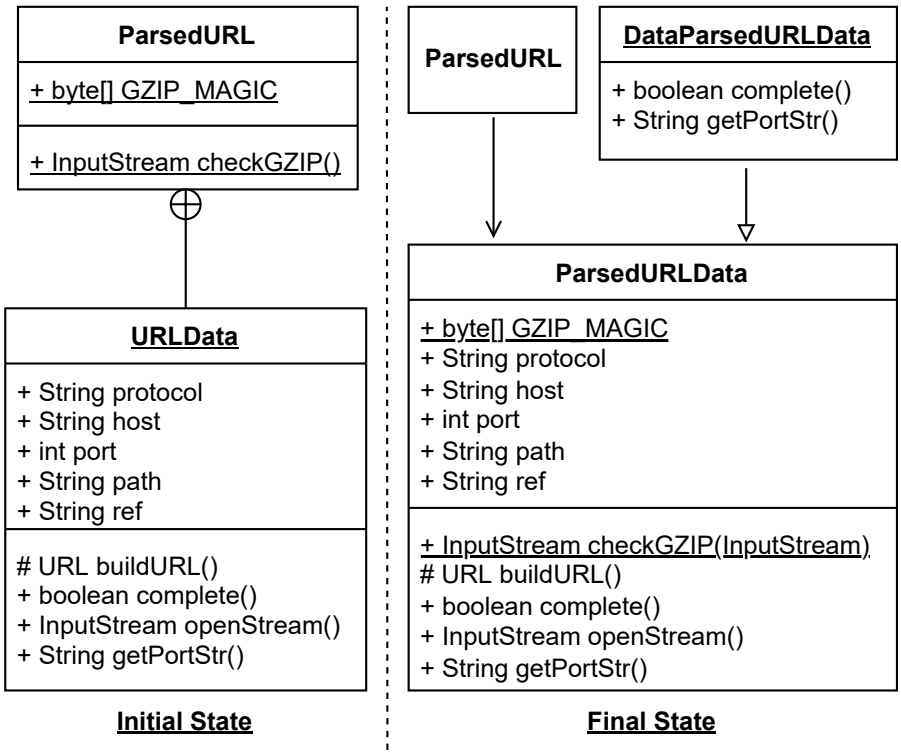

Fig. 3: Graphical representation of the *Extract Superclass* refactoring applied in revision `e28370d2` in project BATIK.

also imply a degradation of encapsulation, further motivating the importance of resolving it early. Curiously, an increase in security remediation effort also happens with *Extract Interface* refactoring. Despite both refactoring operations aiming to simplify the hierarchical organization of a project, they should be made with caution as they have a high risk of increasing the remediation effort. Let us consider an example in project BATIK at the revision `e28370d2`, also shown in Figure 3. The commit applied a series of refactorings to reorganize some hierarchical structures in the `util` package—as also stated in one paragraph of the full commit description: *'[...] Cleaned up, made easier to extend and pulled several inner class out of `ParsedURL` [...]'*. In particular, the class `URLData` was first promoted from a `static` nested class to a first-level `public` class, renamed to `ParsedURLData` and then reorganized to have a new subclass called `DataParsedURLData`—hence, an *Extract Superclass* was applied. In the end, the `ParsedURL` class was streamlined to favor the new extension. In applying these changes, the former nested class had five `public` attributes left unchanged when the class was made `public`. This caused SONARQUBE to recognize a violation to rule S2386 for the five attributes, as now they can be

freely modified by an external client program. In this case, we observe that the refactoring alone is not the direct cause of the violation, but the way it was applied led to the creation of extra `public static` fields. To conclude, *Extract Superclass* and *Extract Interface* refactorings should be applied with particular care as they have also been seen to disrupt all the security metrics ($\mathbf{RQ}_1$).

Another exception occurs with *Extract & Move Method* refactoring, that (1) negatively impacts rule S2647, i.e., *'Basic authentication should not be used'* and (2) increases the chances of rule S5547, i.e., *'Cipher algorithms should be robust'* being removed—as it can be observed from its high OR. The individual refactoring operations, i.e., *Extract Method* and *Move Method*, do not appear to be connected with these rules in any form, while their combination has observable effects. Analyzing this case further, we could not identify specific reasons why the combination of multiple refactorings has a higher impact than individual refactoring types, yet we can suppose that our results represent a reflection of the number of changes applied, i.e., more changes affect security more than individual ones. Nonetheless, the effect of refactoring sequences is something that might be worth further analyzing in future work.

As in $\mathbf{RQ}_1$, it is worth noting that the results were achieved while controlling for several confounding factors. Similarly to the previous discussion, the confounding factors are generally not statistically significant in any model, i.e., they are not correlated with the increase or decrease of security-related technical debt. Likewise, the projects turned out to be significant, somehow confirming that there exist some project-specific attributes that might influence the security technical debt.

Main findings for $\mathbf{RQ}_2$

While most of the refactoring types do not significantly impact security-related violations, we identified some operations concerning restructuring class hierarchies, i.e., *Extract Superclass*, and *Extract Interface*, that are statistically related to an increase of security violations, implying that they should be implemented with caution to avoid introducing security threats. In the end, refactoring is weakly connected to the violations detected by SonarQube, and other influencing factors should be analyzed.

3.3 $\mathbf{RQ}_3$. To what extent do refactoring commits contribute to the introduction of real software vulnerabilities?

Our third research question investigated the relationship between refactoring and the introduction of known vulnerabilities reported in the National Vulnerability Database (NVD).

It is worth recalling that for this research question, we focused on nine of the projects considered in the study (see Table 1). This subset of projects is affected by 26 known vulnerabilities, i.e., 26 different CVE records, whereas

Table 8: The main descriptive statistics pertaining to the unique VCCs of the nine projects appearing in NVD. $N = 103$.

Refactoring	Total	Min	Med	Max	Mean	Std. Dev.
Package-Level						
Move Package	0	0	0	0	0.000	0.000
Class-Level						
Extract Superclass	5	0	0	1	0.049	0.216
Extract Interface	1	0	0	1	0.010	0.099
Move Class	7	0	0	3	0.068	0.377
Method-Level						
Extract Method	24	0	0	6	0.233	0.819
Inline Method	3	0	0	2	0.029	0.220
Move Method	53	0	0	40	0.515	3.983
Extract & Move Method	14	0	0	3	0.136	0.465
Move & Inline Method	2	0	0	1	0.019	0.139
Pull Up Method	251	0	0	231	2.437	22.785
Push Down Method	2	0	0	1	0.019	0.139
Attribute-Level						
Move Attribute	17	0	0	14	0.165	1.387
Pull Up Attribute	126	0	0	81	1.223	9.045
Push Down Attribute	0	0	0	0	0.000	0.000

the number of distinct VCCs was 103—there were some cases of commits contributing to more than one vulnerability. Table 8 reports the descriptive statistics of the distribution of refactoring operations, grouped by type, in such VCCs. In the first place, our results showed that the number of VCCs with at least one refactoring was 34, i.e., 33.01% of the VCCs contained at least one instance of a refactoring operation. While this seems to indicate that refactoring might have a connection with the introduction of vulnerabilities, a closer look indicates a lack of causal relationship between the refactoring activities performed by developers and the introduction of vulnerabilities, i.e., the fact that refactoring is performed does not imply that it is the root cause of the vulnerability introduction.

A more in-depth analysis reveals that the refactoring types that occurred the most in the VCCs were *Pull Up Method* and *Pull Up Attribute* with 251 and 126 instances, respectively. Both refactoring types deal with generalization, hinting that complex restructuring activity (i.e., modifying hierarchies) are often present when vulnerabilities are introduced—this is partially in line with the results observed in the context of $\mathbf{RQ}_2$.

Nonetheless, we also observed their very high standard deviation values that, combined with much lower mean values, imply that the distribution of these refactoring types across the commits is "irregular"—i.e., there are commits with a considerable number of *Pull Up Method* and *Pull Up Attribute* and commits without any of them. For instance, the Jenkins's commit `70c10658` has over 200 instances of *Pull Up Method* and over 80 of *Pull Up Attribute*. Such a commit touches over 300 different files and represents a crucial commit for the project as it marks the moment when Jenkins was forked from Hudson (its original project). This suggests that some projects, like Jenkins, are

characterized by large and poorly cohesive commits, which have higher chances to touch critical parts of the code, possibly introducing defects and security flaws. Hence, a "chaotic" development process might be the actual reason behind the introduction of vulnerabilities—as part of our future research agenda on the matter, we plan to investigate this aspect further.

From a different point of view, the nine NVD projects had a total of 7,708 refactoring commits (i.e., commits with at least one refactoring instance), 34 of which contributed to the introduction of a vulnerability, accounting for 0.44% of the total. This further supports the fact that refactoring alone is not the main responsible for the introduction of vulnerabilities, but rather a co-occurring phenomenon that, in some cases, might worsen the situation, especially when touching several components.

To assess whether the number of a specific refactoring type was statistically significant, we run a one-tailed Mann-Whitney U test [71] for each refactoring type on both the sets (i.e., the refactoring commits contributing to vulnerabilities versus those that did not contribute), for a total of 14 test runs. We discovered that the distribution of *Extract Superclass* and *Extract & Move Method* instances for VCCs is significantly higher than the distribution for non-VCCs. ($p < 0.05$). At the same time, Cohen's d [72] is lower than 0.2, indicating a very small effect size. In other words, *Extract Superclass* and *Extract & Move Method* occurs more often in VCCs, but still in a limited way. *Extract Method* and *Move Method* refactorings do not appear to show any connection with VCCs, hence suggesting that basic refactorings touching few code components are less likely to contribute to the emergence of a vulnerability. On the contrary, only for *Pull Up Method* and *Pull Up Attribute* the effect size appeared large ($d > 1.2$), but without any statistically significant difference highlighted by the Mann-Whitney U test. The full results of such tests are reported in our online appendix [28].

```
44 public class ConversationActivity extends XmppActivity implements
   OnAccountUpdate, OnConversationUpdate, OnRosterUpdate {

   @@ -612,6 +569,6 @@ public void onClick(View v) {
612        getActivity().invalidateOptionsMenu();
613        updateChatMsgHint();
614        if (!activity.shouldPaneBeOpen()) {
615          activity.xmppConnectionService.markRead(conversation, true);
616          activity.updateConversationList();
617        }

   @@ -888,6 +845,10 @@
888      }
889    }
890
891    public void setText(String text) {
892      this.pastedText = text;
893    }
```

```
46 public class ConversationActivity extends XmppActivity implements
   OnAccountUpdate, OnConversationUpdate, OnRosterUpdate {

569        getActivity().invalidateOptionsMenu();
570        updateChatMsgHint();
571        if (!activity.isConversationsOverviewVisable() ||
   !activity.isConversationsOverviewHideable()) {
572          activity.xmppConnectionService.markRead(conversation, true);
573          activity.updateConversationList();
574        }

845      }
846    }
847
848    public void appendText(String text) {
849      String previous = this.mEditMessage.getText().toString();
850      if (previous.length() != 0 && !previous.endsWith(" ")) {
851        text = " " + text;
852      }
853      this.mEditMessage.append(text);
854    }
```

Fig. 4: Part of the diff of the commit `e45d7bda` of Conversations. The focus is given on the root cause behind the vulnerability CVE-2018-18467, i.e. caused by the addition of an incomplete `appendText()` method during the refactoring.

Going more in-depth, let us consider the example reported in Figure 4. It shows the diff of the commit `e45d7bda` of the project Conversations, an XMPP client for Android that allows the creation of private chats with other users. The commit message states that a *"UI code refactoring"* was applied. The modification impacted three different files. Four years later, the modification resulted in being one of the causes that led to vulnerability CVE-2018-18467, which allowed an attacker to append a custom text to an existing conversation (with a draft message) by sending an intent from another application. While the commit message suggests code refactoring as the main activity performed, the vulnerability was not due to the refactoring itself, but rather to the addition of the `appendText()` method in the class `ConversationFragment`. This allowed appending any text to an existing conversation without adequately checking if an external application was trying to append a text content to an existing draft message through an `Intent`, leading to the vulnerability described in CVE-2018-18467. In other words, the vulnerability was involuntarily introduced while the committing author was doing some refactoring and code clean-up. In the example, RefactoringMiner only managed to mine two instances of *Inline Method*, which only represent a small part of the total modifications made. Thus, we can conclude that refactoring is often not the direct cause of vulnerabilities but rather a co-occurring phenomenon.

Main findings for RQ_3

Our results indicate the absence of a clear cause-effect relationship between refactoring and vulnerability-contributing commits. At the same time, we also observed that some refactoring operations, such as *Extract Superclass* and *Extract & Move Method* co-occur often in commits where vulnerabilities are introduced.

4 Discussion and Implications

This section further discusses the main results achieved in our study and reports their implications for researchers and practitioners.

4.1 Discussion: Connecting the Dots

The results of our three research questions allowed us to quantify the role of refactoring on three critical aspects of software security, such as its impact on security metrics, technical debt, and introduction of known vulnerabilities. Moreover, the statistical analyses conducted enable a more general and conclusive discussion of the initial hypotheses formulated in Table 2.

By summing all up, we can provide three main insights. First of all, when looking at the big picture, we can conclude that refactoring has only a *limited effect* on software security. Most of the refactoring operations considered in the study do not lead any security indicators to vary consistently and/or

significantly. As we learned from $\mathbf{RQ}_1$, it is indeed possible that some refactoring operations may influence security metrics depending on how they are applied, while they rarely have an impact on technical debt ($\mathbf{RQ}_2$) and introduction of known vulnerabilities ($\mathbf{RQ}_3$); similarly, it may happen that a change accompanied by refactoring can contribute to a vulnerability without affecting any security metrics or increasing the technical debt value. This is the case of the example shown in Figure 4 in which neither SURFACE nor SONARQUBE detected any difference between the commit containing the refactoring (`e45d7bda`) and its predecessor (i.e., the Δ was mapped to the "Stable" category). Hence, these results partially contradict the preliminary findings reported by Adib et al. [15]: when studying the matter on a larger scale, it comes out that most refactoring operations do not directly impact the security of software systems but are rather co-occurring phenomena.

However, some exceptions have been observed, especially when considering security technical debt. According to our results, the *Extract Interface* refactoring provides a significant increase in security-related technical debt but might have positive effects on some security metrics, e.g., reducing the number of classified methods (CM metric). This result supports and further stimulates the research efforts on the construction of automated refactoring recommenders that might balance quality improvements and security threats, as initiated by Abid et al. [15].

To broaden the scope of the discussion, our overall findings do not match with the results previously obtained when studying the relation between refactoring and defects [19, 16]. In particular, this is the case of the refactoring types dealing with the generalization: while Bavota et al. [19] and Di Penta et al. [16] found these operations to be sometimes defect-inducing, we discovered that they can instead provide some benefits to security aspects connected to attribute encapsulation. We see two main points here. On the one hand, these differences corroborate the conclusions drawn by previous researchers on the need of considering and treating vulnerabilities differently from defects [75, 76, 77, 78, 79, 80]. On the other hand, our results indicate that the same refactoring can have multiple, contrasting effects on code quality and dependability.

4.2 Implications of the Study

The results of our study provide us with several actionable items and implications for both researchers and practitioners that we discuss in the following.

Novel Refactoring Optimization Techniques. According to our results, refactoring is generally not connected to software vulnerabilities. However, we pointed out that refactoring operations dealing with generalization can contribute to the improvement of software systems' security profile under certain perspectives. By connecting the previous research on the effect of refactoring on defects introduction with the results of our study,

we could conclude that the definition of novel strategies that recommend refactoring operations—while minimizing the negative impact on source code attributes—should be devised and further investigated. In particular, the key example is represented by search-based refactoring recommendations [81, 82], where search-based algorithms are used to recommend developers the best refactoring operation (or sequence of operations) to apply based on the potential impact that such a refactoring may have on various properties of source code. These recommenders might be potentially enhanced by means of the addition of further security-related objective functions so that they could recommend refactoring operations that optimize the compromise between quality and security metrics/technical debt. For instance, let us consider the case of *Extract Superclass*, which we found to appear among the most disruptive refactorings for security according to our results. The refactoring consists of finding a subset of methods of a class that can be extracted in order to create a new superclass. As such, there are multiple ways to perform the refactoring based on how the subset of methods to extract is identified. An *Extract Superclass* refactoring recommender might use, as an objective function, a weighted combination of quality and security metrics so that it can identify the subset of methods to extract that optimize both quality and security. Similarly, multi-objective search algorithms might be used to solve the problem, for instance by combining information coming from quality metrics, security metrics, and technical debt. Some preliminary studies on these aspects have been recently published [15], yet we believe that further studies are needed, especially concerning the granularity of the recommendations. Indeed, our $\mathbf{RQ}_3$ shows that developers would benefit from just-in-time solutions that can provide advice while committing new changes to software repositories.

Exploiting Security Variations to Drive Refactoring. The results coming from $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$ also pointed out the existence of refactoring operations having high correlations with both increase or decrease of security metrics and technical debt. In these cases, our findings suggest that the positive or negative effect on security is due to the specific operation performed when refactoring code. In a real-case scenario, these results may be exploited to devise automated mechanisms that alert developers of the potential effects of refactoring on security. As an example, we may envision the definition of novel bots/conversational agents [83] that monitor the development and drive the developer toward the application of an operation that has higher chances to improve security metrics or reduce security technical debt when recognizing he/she is applying a refactoring operation. The research in this respect is rapidly gaining interest [84, 85, 86], though actionable solutions are still not widely spread, and so representing a potentially interesting use case.

Refactoring Verification and Validation. As a complimentary discussion of the previous one, we can foresee two main implications for the testing community. First and foremost, the importance of having robust verification and

validation techniques is further corroborated by our study. Practitioners and security managers can indeed exploit our findings to put in place additional preventive mechanisms aimed at verifying the outcome of each modification, possibly improving both the code review process [39], e.g., by integrating stricter security checks when refactoring operations are applied, and the regression testing activities [87] of their systems. Secondly, our results shed light on the need for more research on techniques to verify the correctness of refactoring operations. This is an overly neglected angle of the refactoring process [88] that has been only tangentially touched by the research community in the past [89, 90]. We hope that our investigation would stimulate research on this topic.

Homogenizing Refactoring Operations. As noticed in $\mathbf{RQ}_1$, some refactoring operations, e.g., *Extract Superclass*, tend to have different effects for security depending on how they are applied. This finding—which we believe would deserve further attention—possibly suggests that practitioners approach refactoring in different manners, perhaps because of their different expertise or level of knowledge on the classes subject to refactoring. As such, they could benefit from automated solutions that can recommend *how* to apply the refactoring, namely what are the steps that may lead to the safe improvement of source code quality and homogenize the refactoring process toward the definition of standard guidelines that might favor both newcomers and developers with limited knowledge on security.

The Link between Composite and Elementary Refactorings. In this empirical study we investigated the effect of two composite refactoring operations, namely *Extract & Move Method* and *Move & Inline Method*. We suspected that they might behave differently from the isolated application of the basic refactoring operations they are composed of—i.e., *Extract Method*, *Move Method*, and *Inline Method*. The results of $\mathbf{RQ}_1$ show that composite refactoring operations appear to behave as if they are a "mixture" of their basic operations; conversely, in $\mathbf{RQ}_2$ we observed that they have some effects on a restricted subset of SonarQube violations, while their basic operations do not. Similarly, in $\mathbf{RQ}_3$ *Extract & Move Method* tends to occur more often in vulnerability-contributing commits than individual *Extract Method* and *Move Method* refactorings. Based on these results, we could not outline a precise trend regarding composite refactorings. In any case, the number of instances in our dataset was limited, hence demanding further in-depth investigations with a larger number of observations to derive more precise conclusions on how these composite refactoring operations are connected to their individual refactoring operations.

Value of The Currently Available Security Metrics. When collecting the data required to address $\mathbf{RQ}_1$, we observed that the security metrics previously proposed in the literature [15, 46, 47] capture similar aspects, being all computed based on the number of security-sensitive attributes that a class exposes. We consider it a limitation that does not enable a comprehensive analysis of the source code's security profile. As such, a side outcome of

our study suggests that more effort should be invested in the definition of novel security metrics that may adequately complement the analysis of the attributes. This represents a challenge for the software engineering community and researchers in closely related fields, e.g., programming languages, which are called to elicit specific properties that make programming languages more or less prone to security weaknesses.

Refactoring Has A Poor Impact on Security Technical Debt. From the results achieved in the context of $\mathbf{RQ}_2$, we could observe that most refactoring operations do not significantly vary the amount of security-related technical debt. The *Extract Superclass* and *Extract Interface* refactoring types represent exceptions to this discussion, along with composite refactoring operations. As such, we can claim that refactoring is mostly safe with respect to security technical debt, yet verification and validation mechanisms might represent useful additions to assess the refactored code against security regressions.

Software Vulnerabilities: A Social Perspective? The results given by our statistical modeling exercise revealed that, in most cases, the projects themselves turn out to significantly influence the increase/decrease of security-related metrics and technical debt. While this aspect deserves ad-hoc investigations to better understand the underlying reasons leading to these findings, our study seems to suggest that there exist specific properties or standards implemented within those projects that have effects on software dependability. In other words, our results seem to be in line with recent studies uncovering relations between developer's collaboration/coordination—elaborated and controlled through the definition of development contribution guidelines [74] and code of conducts [38]—and the implications they have for software quality [91, 92]. In this sense, our results can serve as a base for investigations into the role of social aspects on vulnerabilities.

5 Threats to Validity

Several factors might have biased our results. This section discusses them and reports the mitigation strategies we employed.

Construct Validity. The subjects of our study were the commits having refactoring operations, that we could detect using the tool RefactoringMiner [24]. The main threat associated with this granularity level is the impossibility to isolate the refactored code elements and study how their security profile has changed. Despite the fact that RefactoringMiner allows the identification of the refactored code regions, we still had trouble in selecting a reasonable set of metrics capturing the security profiles at such a granularity level. In other words, there are no metrics that can measure the security of partial code snippets: the minimum unit of work is the file/class. Nevertheless, we strove for addressing at our best this issue by removing the amount of noise from refactoring commits—whenever the metrics and tools allowed.

Specifically, the computation of the security metrics ($\mathbf{RQ}_1$), the number of violations ($\mathbf{RQ}_2$), and the confounding variables (both $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$) did not involve the files not subject to any of the refactoring operations occurred in a commit. Unfortunately, we could not do the same for the security remediation effort metric ($\mathbf{RQ}_2$) as the tool SonarQube is only able to compute it at the entire project's snapshot level. In spite of everything, this mitigation mechanism still does not exclude any form of changes unrelated to refactorings. Currently, this is the best possible solution to the best of our knowledge.

In the context of $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$, we employed automated tools to compute security metrics and technical debt. As for the security metrics, we reimplemented the tool by Abid et al. [15] as it was not publicly available. When developing Surface, we followed the exact steps reported in [15], other than conducting follow-up automated and manual testing sessions to assess the results produced by the tool. For the sake of verifiability, we made Surface publicly available in our online appendix [28]. Among the technical debt detectors available in the literature, the selection of SonarQube was driven by the results reported by Saarimaki et al. [50], who showed that it is accurate when considering security violations. Moreover, these tools were supported by the libraries PyDriller [62] and Lizard to facilitate the recovery of the change history and the computation of the confounding variables (LOC, WCM, and code churn), respectively. Both are widely applied in several software repository mining studies.

We expressed the security-related metrics for the commits by aggregating the deltas we computed on all the files directly involved in refactorings. In particular, the metrics CA (Classified Attributes) and CM (Classified Methods) were summed, while the rest of the metrics were averaged. CA and CM count the number of security-sensitive code components (attributes or methods), so the sum suits well to count the amount of changed security-sensitive code components within the commit. On the contrary, all the other security metrics ranged between 0 and 1, expressing "no exposure" to "maximal exposure", respectively. Despite the existence of other aggregators, such as the median, we opted to use the average as it well summarizes the change in the exposure levels of all the refactored files without reducing the effect of outliers (i.e., sharp changes in the security metrics).

As for $\mathbf{RQ}_3$, our results might have been affected by the erroneous identification of vulnerability-fixing and vulnerability-contributing commits. In the first case, we mined the fixing commits from the references reported in the CVE records description in the National Vulnerability Database (NVD). Despite being considered a reliable source of information that is continuously monitored and updated, we cannot exclude the case in which the CVE record fails at reporting the entire set of patches—indeed, an insufficient set of fixing commits would have reduced the amount of contributing commits our algorithm fetched. In the second case, we employed a set of heuristics built on top of SZZ [34] to recover the VCCs. While the performance of the algorithm has been criticized in the past [93], a recent study [94] has shown that (i) the performance of SZZ depends on the dataset to which it is applied and (ii)

the original version of SZZ is the one providing the best performance, overall. Moreover, it has been seen as one of the best possible strategies for recovering VCCs. For this reason, we were careful to adopt all possible recommended precautions to greatly reduce the amount of noise and improve the precision, e.g., ignoring irrelevant files, blaming the context of blocks of new code, etc. To be even more confident about the suitability of our VCCs mining algorithm, we manually validated its results, observing a precision of 71%, which we considered acceptable for our purposes. Lastly, our strategy is also robust to most cases of files renamings. As a matter of fact, the `git-blame` functionality can automatically detect file renamings when traversing the project's history, further reducing the risk of blaming wrong commits.

Internal Validity. When building statistical models in $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$, we selected three confounding factors, i.e., LOC, WMC, and code churn, to control our findings for aspects that might have explained the (in)stability of security metrics and violations better than the number of instances of refactoring operations. We acknowledge the existence of additional factors that were not considered in our study, and, as such, replications of our work would be desirable. Nonetheless, our manual follow-up analysis (see Section 4.1) had the goal of further investigating the underlying reasons behind the relation between refactoring and vulnerabilities, possibly mitigating threats to internal validity and also explaining the role of confounding factors on our results.

Different implementations of refactoring operations might affect the level of security of source code differently or may even represent explicit compromises between quality and security made by a developer. For example, a *Pull Up Attribute* refactoring typically leads to a visibility change of a `private` attribute: this might be either performed by modifying the visibility into `protected` or `public` so that the attribute can be accessible by child classes. While the `protected` visibility would be essential to apply the refactoring, the `public` visibility might potentially induce unnecessary risks for security—unless developers consciously opt for this choice and favor it because of other contextual factors or requirements. In this respect, it is worth remarking that our empirical study does not aim at questioning the way developers may apply refactoring, but rather what effect refactoring types may have on the security profile of source code. Furthermore, the specific design decisions taken by a developer when performing refactoring cannot be automatically detected through the refactoring mining tools currently available. Therefore, we encourage replications of our study conducted with different research methods, e.g., through controlled experiments that verify how the refactoring choices done by developers impact security.

Conclusion Validity. Concerning the relation between treatment and outcome, a threat is related to the statistical methods adopted to address our **RQ**s. In $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$, we opted for a Multinomial Log-Linear statistical model [33] as our problem was a multiclass problem involving both categorical and continuous independent variables. In addition, it allowed us to interpret the results from various perspectives, i.e., by considering both statistical codes

and odds ratios. Before interpreting the results, we also verified the normality of the independent variable distributions through the Anderson-Darling normality test [64] before computing the Spearman's rank-correlation coefficients [65] to identify pairs of correlated independent variables that might lead to multicollinearity.

External Validity. Our study targeted 39 projects from the *Technical Debt Dataset* [35] involving 7,708 commits containing refactorings. While almost all the systems belong to the Apache Software Foundation, they were originally selected to meet guidelines that ensure diversity and representativeness [36, 37]. We cannot exclude that different results could be obtained when considering systems of other ecosystems developed using different programming languages and with different maturity levels. In addition, it is worth remarking that $\mathbf{RQ}_3$ could only target nine of those projects, namely the ones connected to the NVD dataset of known vulnerabilities. Replications targeting a larger set of projects would be, therefore, desirable. In any case, in our online appendix [28], we made available the data and scripts to favor researchers interested in replicating our study in other contexts.

6 Related Work

The impact of refactoring on source code dependability has been explored from different perspectives, which we overview herein.

6.1 Impact of Refactoring on Software Quality

Many studies have investigated the impact of refactoring on software quality either directly or from the perspective of defect proneness, change proneness, or code smells.

Bavota et al. [9] mined the history of 63 releases of Java Open Source Projects (OSPs) to investigate refactoring operations on code components, which indicate a need for refactoring through indicators such as metrics and smells. They concluded that most refactoring operations take place on code with no quality metrics indication for the need to refactor, and although 40% of refactoring operations were performed on smelly code, only 7% removed the smells. Their findings were corroborated by Yoshida et al. [95] who revisited the relationship between code smells and refactoring by using the same refactoring dataset by Bavota et al. [9]. Cedrim et al. [96] had similar findings when they analyzed more than 16K+ refactoring instances from 23 OSPs to investigate whether refactoring reduces the code smell density. They reported that even though almost 80% of refactorings touched smelly code, 57% of refactorings did not impact them, and roughly 10% of them removed the smells while 33% introduced new smells. Tufano et al. [97] conducted an empirical study on 200 projects from the Android, Apache, and Eclipse ecosystems to investigate, among other aspects, whether developers' actions, e.g., refactoring, resolve

smells. They reported that only a low number (9%) of code smells are removed following refactoring operations.

Palomba et al. [17] investigated the relationship between refactoring operations and code changes (namely fault repairing modification, general maintenance modification, and feature introduction modification) by analyzing the dataset by Bavota et al. [9]. They concluded that code duplication and Self-Admitted Technical Debt (SATD) are the main reasons behind refactoring instances, and refactoring also helps increase code readability. The impact of refactoring on code readability was the subject of the study by Sellitto et al. [25], who partially confirmed previous findings, showing that refactoring can also negatively impact code readability metrics.

Kim et al. [98] investigated the refactoring benefits and challenges at Microsoft by conducting a survey, semi-structured interviews, and historical data analysis. They found that refactoring is beneficial, leading to reduced inter-module dependencies and post-release defects. An empirical study was conducted by Bavota et al. [19] to investigate the impact of refactoring on defects. They found that generally, refactoring instances do not induce defects. However, in specific cases, some specific refactoring types (e.g., Pull Up Method and Extract Subclass) tend to introduce defects in code.

6.2 Impact of Refactoring on Software Security

Mumtaz et al. [99] investigated whether removing code smells through refactoring resulted in improved security for a system. They conducted a study to identify a subset of code smells and calculated security metrics on five systems. Then they applied refactoring to remove the smells and then re-calculated the same metrics as before. They concluded that generally, refactoring improved the quality of the studied systems from a security standpoint. Ghaith et al. [100] were interested in finding out whether automated search-based refactoring improved software security. They achieved an improvement of 15% in the metrics of industrial software after applying search-based refactoring. However, their study is based on a small project, and the results are not generalizable.

An empirical study was conducted by Abid et al. [15] to determine the relationship between quality and security and the impact of refactoring types on security. The results of the study were used to implement a tool, which was then evaluated on OSPs. They concluded that their tool improved the security of the systems with little impact on the quality. They further validated their results by conducting a survey with practitioners. Similarly, Alshammari et al. [101] assessed the impact of refactoring at the design level on security using a case study. Their findings indicate that about 20 refactoring rules improve security, 12 rules made security worse, and four rules had no impact on security. In a follow-up study [73], the authors evaluated the impact of refactoring on information security using a case study. 8 out of 16 refactoring rules used improved the software's security while the remaining made it worse. Again, both

studies being focused on one case study cannot be generalized. Maruyama et al. [102] proposed a tool, implemented as an Eclipse plug-in, to help developers assess the impact of their refactoring operations on software vulnerabilities during software implementation. Currently, the tool supports only two refactoring types, namely, *Pull Up Method* and *Push Down Method*, and measure security using access levels (private, public, protected, and default) of fields. A downgrade in the access level signifies that the software becomes more vulnerable. However, they evaluated the tool using an artificial experiment (on one version of Eclipse) and not on real software.

6.3 Impact of Refactoring on Security-Related Technical Debt

Refactoring has been recognized in many studies as one of the most common ways to manage technical debt [103, 104, 105]. In this subsection, we review some studies to understand the impact of refactoring on security debt (TD).

Zabardast et al. [106] investigated the impact of various software development activities, including refactoring on TD by analyzing 2K+ commits in a large industrial project. Their empirical study shows that refactoring removes 22% of TD but introduces an additional 22% TD. However, in most cases, refactoring did not impact TD. Search-based automated refactoring using four different approaches was used by Mohan et al. [107] to determine the impact of refactoring on TD, among other aspects of development, on six OSPs. They concluded that automated refactoring help decrease TD in software. However, these studies do not focus on security-related TD. Similarly, there are some studies on security smells, which are symptoms in the code that signals the prospect of a security vulnerability [108]. Such studies investigate whether the security smells have an impact on vulnerabilities and are conducted in specific domains [108, 109].

6.4 Reflecting on Previous Work and Our Contribution

To summarize, the studies that investigate the impact of refactoring on software quality have mixed results. Some reported a positive impact whereas others concluded that refactoring increased TD or had no impact on it. Most existing studies focus on the impact of refactoring on software quality but very few investigate TD specifically. Similarly, the studies which investigate the impact of refactoring on software security do not include security-related technical debt. The studies are also limited, often focusing on one software system, thereby making their results not generalizable. Despite refactoring being commonly used to reduce technical debt, most existing research focuses on code smell as an indication of debt, thereby explaining the lack of refactoring studies that focus on technical debt directly. Similarly, security smells have been investigated in the context of vulnerabilities but not refactoring. Abid et al. [15] conducted a preliminary study to investigate the impact of refactoring

types on security but, to the best of our knowledge, investigating the actual impact of refactoring on technical debt from a security standpoint has not been studied and therefore represents a premier of our research.

7 Conclusion

The potential adverse effects of refactoring on software dependability have been previously assessed concerning its relation to software defects [19, 16]. In this study, we went a step forward by considering the extent to which refactoring affects software security. We have conducted a three-level analysis that considered the effects of refactoring on (i) security metrics, (ii) security-related technical debt, and (iii) contribution to the introduction of known vulnerabilities. Our study had a primarily quantitative connotation where we employed statistical methods on a set of 39 open-source projects. Yet, we conducted additional manual analyses to extract qualitative insights and possible motivations explaining the statistical findings. The core results of the study reported that refactoring has a limited impact on security. Nevertheless, some exceptions indicate that some particular types of refactoring operations might lead to significant variations of software systems' security profiles under different perspectives. Particularly interesting, in this respect, was the case of refactoring operations dealing with the generalization that appeared to disrupt the source code security.

Based on our findings, we identified several open issues and challenges for researchers, especially related to the lack of automated mechanisms to balance multiple dependability attributes. These outcomes represent our future research agenda, which is focused on the definition of novel just-in-time vulnerability detectors, technical debt linters, and testing methods to verify the presence and exploitability of software vulnerabilities. Additionally, we plan to extend the study by considering a more comprehensive range of software projects, refactoring operations (e.g., "big" or architectural refactoring [1]), and security-related indicators (e.g., security smells [108]), other than triangulating our findings with different research methods, e.g., through controlled studies able to reveal how different refactoring implementations may lead to a variation of software security indicators.

Acknowledgement

The authors would like to thank the associated handling editor and the anonymous reviewers for their insightful suggestions and feedback, which were instrumental in improving the quality of our manuscript. Fabio and Zadia gratefully acknowledge the support of the Swiss National Science Foundation (SNSF) through the SNF Project No. PZ00P2_186090 (TED) and the Natural Sciences and Engineering Research Council of Canada (RGPIN-2021-04232 and DGECR-2021-00283), respectively. This work has been partially supported by

the EMELIOT national research project, funded by the MUR under the PRIN 2020 program (Contract 2020W3A5FY). This work was partially supported by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU.

Declarations

Funding and/or Conflicts of interests/Competing interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data Availability Statement

The datasets built during the current study, plus the scripts used to analyze and generate the data, are available in the FigShare repository: `https://figshare.com/articles/online_resource/Rubbing_Salt_in_the_Wound_A_Large-Scale_Investigation_into_the_Effects_of_Refactoring_on_Vulnerabilities/14483787/1`.

References

1. F. Martin and B. Kent, "Refactoring: Improving the design of existing code," *Addison-Wesley Longman Publishing Co., Inc.*, 1999.
2. J. Al Dallal and A. Abdin, "Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 44, no. 1, pp. 44–69, 2017.
3. M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, vol. 108, pp. 115–138, 2019.
4. E. V. de Paulo Sobrinho, A. De Lucia, and M. de Almeida Maia, "A systematic literature review on bad smells—5 w's: which, when, what, who, where," *IEEE Transactions on Software Engineering*, 2018.
5. T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on software engineering*, vol. 30, no. 2, pp. 126–139, 2004.
6. G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: an improved method and its evaluation," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1617–1664, 2014.
7. R. Terra, M. T. Valente, S. Miranda, and V. Sales, "Jmove: A novel heuristic and tool to detect move method refactoring opportunities," *Journal of Systems and Software*, vol. 138, pp. 19–36, 2018.

8. N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.
9. G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1–14, 2015.
10. D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *24th acm sigsoft international symposium on foundations of software engineering*, 2016, pp. 858–870.
11. E. Murphy-Hill and A. P. Black, "Breaking the barriers to successful refactoring: observations and tools for extract method," in *International conference on Software engineering*, 2008, pp. 421–430.
12. M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoringchallenges and benefits at microsoft," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 633–649, 2014.
13. T. Sharma, G. Suryanarayana, and G. Samarthyam, "Challenges to and solutions for refactoring adoption: An industrial perspective," *IEEE Software*, vol. 32, no. 6, pp. 44–51, 2015.
14. C. Vassallo, F. Palomba, and H. C. Gall, "Continuous refactoring in ci: A preliminary study on the perceived advantages and barriers," in *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 564–568.
15. C. Abid, M. Kessentini, V. Alizadeh, M. Dhouadi, and R. Kazman, "How does refactoring impact security when improving quality? a security-aware refactoring approach," *IEEE Transactions on Software Engineering*, 2020.
16. M. Di Penta, G. Bavota, and F. Zampetti, "On the relationship between refactoring actions and bugs: a differentiated replication," in *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 556–567.
17. F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia, "An exploratory study on the relationship between changes and refactoring," in *25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 176–185.
18. K. Stroggylos and D. Spinellis, "Refactoring–does it improve software quality?" in *International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)*. IEEE, 2007, pp. 10–10.
19. G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *12th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2012, pp. 104–113.
20. G. McGraw, "Software security," *IEEE Security & Privacy*, vol. 2, no. 2, pp. 80–83, 2004.
21. V. Alizadeh, M. Kessentini, M. W. Mkaouer, M. Ocinneide, A. Ouni, and Y. Cai, "An interactive and dynamic search-based approach to software refactoring recommendations," *IEEE Transactions on Software En-*

gineering, vol. 46, no. 9, pp. 932–961, 2018.
22. P. K. Goyal and G. Joshi, "Qmood metric sets to assess quality of java program," in *International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*. IEEE, 2014, pp. 520–533.
23. B. Alshammari, C. Fidge, and D. Corney, "Security metrics for object-oriented class designs," in *International Conference on Quality Software*. IEEE, 2009, pp. 11–20.
24. N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *40th International Conference on Software Engineering*, ser. ICSE '18, 2018, pp. 483–494.
25. G. Sellitto, E. Iannone, Z. Codabux, V. Lenarduzzi, A. De Lucia, F. Palomba, and F. Ferrucci, "Toward understanding the impact of refactoring on program comprehension," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022, pp. 731–742.
26. H. J. Adèr, *Advising on research methods: A consultant's companion*. Johannes van Kessel Publishing., 2008.
27. "National vulnerability database," https://nvd.nist.gov/.
28. E. Iannone, Z. Codabux, V. Lenarduzzi, A. De Lucia, and F. Palomba, "Rubbing salt in the wound? a large-scale investigation into the effects of refactoring on security," Mar 2022. [Online]. Available: https://figshare.com/articles/online_resource/Rubbing_Salt_in_the_Wound_A_Large-Scale_Investigation_into_the_Effects_of_Refactoring_on_Vulnerabilities/14483787/1
29. R. Per and H. Martin, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Softw. Engg.*, vol. 14, no. 2, pp. 131–164, 2009.
30. P. Ralph, N. bin Ali, S. Baltes, D. Bianculli, J. Diaz, Y. Dittrich, N. Ernst, M. Felderer, R. Feldt, A. Filieri, B. B. N. de França, C. A. Furia, G. Gay, N. Gold, D. Graziotin, P. He, R. Hoda, N. Juristo, B. Kitchenham, V. Lenarduzzi, J. Martínez, J. Melegati, D. Mendez, T. Menzies, J. Molleri, D. Pfahl, R. Robbes, D. Russo, N. Saarimäki, F. Sarro, D. Taibi, J. Siegmund, D. Spinellis, M. Staron, K. Stol, M.-A. Storey, D. Taibi, D. Tamburri, M. Torchiano, C. Treude, B. Turhan, X. Wang, and S. Vegas, "Empirical standards for software engineering research," 2021.
31. B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the size, cost, and types of technical debt," in *2012 Third International Workshop on Managing Technical Debt (MTD)*. IEEE, 2012, pp. 49–53.
32. S. Sukamolson, "Fundamentals of quantitative research," *Language Institute Chulalongkorn University*, vol. 1, pp. 2–3, 2007.
33. H. Theil, "A multinomial extension of the linear logit model," *International economic review*, vol. 10, no. 3, pp. 251–259, 1969.
34. J. Sliwerski, Z. T., and A. Zeller, "When do changes induce fixes?" in *International Workshop on Mining Software Repositories*, ser. MSR '05, 2005, pp. 1–5.

35. V. Lenarduzzi, N. Saarimäki, and D. Taibi, "The technical debt dataset," in *15th conference on PREdictive Models and data analycs In Software Engineering*, ser. PROMISE '19, 2019, pp. 2 – 11.
36. M. Nagappan, T. Zimmermann, and C. Bird, "Diversity in software engineering research," in *2013 9th joint meeting on foundations of software engineering*, 2013, pp. 466–476.
37. M. Patton, *Qualitative Evaluation and Research Methods*. Newbury Park: Sage, 2002.
38. P. Tourani, B. Adams, and A. Serebrenik, "Code of conduct in open source projects," in *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2017, pp. 24–33.
39. L. Pascarella, D. Spadini, F. Palomba, M. Bruntink, and A. Bacchelli, "Information needs in contemporary code review," *Proceedings of the ACM on Human-Computer Interaction*, vol. 2, no. CSCW, pp. 1–27, 2018.
40. "U.s. nist computer security division," https://www.nist.gov.
41. "Common vulnerabilities and exposures," https://cve.mitre.org/.
42. O. Alhazmi, Y. Malaiya, and I. Ray, "Measuring, analyzing and predicting security vulnerabilities in software systems," *Computers & Security*, vol. 26, no. 3, pp. 219–228, 2007.
43. S. Huang, H. Tang, M. Zhang, and J. Tian, "Text clustering on national vulnerability database," in *International Conference on Computer Engineering and Applications*, vol. 2, 2010, pp. 295–299.
44. S. Zhang, D. Caragea, and X. Ou, "An empirical study on using the national vulnerability database to predict software vulnerabilities," in *International Conference on Database and Expert Systems Applications*, 2011, pp. 217–231.
45. D. Silva, J. Silva, G. J. D. S. Santos, R. Terra, and M. T. O. Valente, "Refdiff 2.0: A multi-language refactoring detection tool," *IEEE Transactions on Software Engineering*, 2020.
46. B. Alshammari, C. Fidge, and D. Corney, "Security metrics for object-oriented designs," in *Australian Software Engineering Conference*. IEEE, 2010, pp. 55–64.
47. A. Agrawal and R. Khan, "Assessing impact of cohesion on security-an object oriented design perspective," *Pensee*, vol. 76, no. 2, 2014.
48. 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.
49. P. Avgeriou, D. Taibi, A. Ampatzoglou, F. Arcelli Fontana, T. Besker, A. Chatzigeorgiou, V. Lenarduzzi, A. Martini, N. Moschou, I. Pigazzini, N. Saarimäki, D. Sas, S. Soares de Toledo, and A. Tsintzira, "An overview and comparison of technical debt measurement tools," *IEEE Software*, 2021.
50. N. Saarimaki, M. T. Baldassarre, V. Lenarduzzi, and S. Romano, "On the accuracy of sonarqube technical debt remediation time," in *Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*.

IEEE, 2019, pp. 317–324.
51. "Cve search tool," https://github.com/cve-search/cve-search.
52. M. H. Kutner, C. J. Nachtsheim, J. Neter, W. Li *et al.*, *Applied linear statistical models*. McGraw-Hill Irwin Boston, 2005, vol. 5.
53. K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai, "The confounding effect of class size on the validity of object-oriented metrics," *IEEE Transactions on Software Engineering*, vol. 27, no. 7, pp. 630–650, 2001.
54. A. G. Koru and H. Liu, "An investigation of the effect of module size on defect prediction using static measures," in *Workshop on Predictor models in software engineering*, 2005, pp. 1–5.
55. H. Zhang, "An investigation of the relationships between lines of code and defects," in *International Conference on Software Maintenance*. IEEE, 2009, pp. 274–283.
56. S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.
57. T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.
58. I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *Journal of Systems Architecture*, vol. 57, no. 3, pp. 294–313, 2011.
59. Y. Shin and L. Williams, "An empirical model to predict security vulnerabilities using code complexity metrics," in *International symposium on Empirical software engineering and measurement*, 2008, pp. 315–317.
60. N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *International conference on Software engineering*, 2005, pp. 284–292.
61. Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE transactions on software engineering*, vol. 37, no. 6, pp. 772–787, 2010.
62. D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. New York, New York, USA: ACM Press, 2018, pp. 908–911. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3236024.3264598
63. J. M. Bland and D. G. Altman, "The odds ratio," *Bmj*, vol. 320, no. 7247, p. 1468, 2000.
64. T. W. Anderson and D. A. Darling, "Asymptotic Theory of Certain "Goodness of Fit" Criteria Based on Stochastic Processes," *The Annals of Mathematical Statistics*, vol. 23, no. 2, pp. 193 – 212, 1952. [Online]. Available: https://doi.org/10.1214/aoms/1177729437
65. C. Spearman, "The proof and measurement of association between two things." 1961.

66. A. Meneely, H. Srinivasan, A. Musa, A. Tejeda, M. Mokary, and B. Spates, "When a patch goes bad: Exploring the properties of vulnerability-contributing commits," in *International Symposium on Empirical Software Engineering and Measurement*, 2013, pp. 65–74.
67. G. Rodríguez-Pérez, G. Robles, and J. M. González-Barahona, "Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm," *Inf. Softw. Technol.*, vol. 99, no. C, p. 164–176, jul 2018. [Online]. Available: https://doi.org/10.1016/j.infsof.2018.03.009
68. H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 426–437. [Online]. Available: https://doi.org/10.1145/2810103.2813604
69. L. Yang, X. Li, and Y. Yu, "Vuldigger: A just-in-time and cost-aware tool for digging vulnerability-contributing changes," in *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, 2017, pp. 1–7.
70. E. Iannone, R. Guadagni, F. Ferrucci, A. De Lucia, and F. Palomba, "The secret life of software vulnerabilities: A large-scale empirical study," *IEEE Transactions on Software Engineering*, pp. 1–1, 2022.
71. H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *Annals of Mathematical Statistics*, vol. 18, pp. 50–60, 1947.
72. J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*. Taylor & Francis, 2013.
73. B. Alshammari, C. Fidge, and D. Corney, "Security assessment of code refactoring rules," in *National Workshop on Information Assurance Research*. VDE, 2012, pp. 1–10.
74. O. Elazhary, M.-A. Storey, N. Ernst, and A. Zaidman, "Do as i do, not as i say: Do contribution guidelines match the github contribution process?" in *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 286–290.
75. G. Canfora, A. Di Sorbo, S. Forootani, A. Pirozzi, and C. A. Visaggio, "Investigating the vulnerability fixing process in oss projects: Peculiarities and challenges," *Computers & Security*, vol. 99, p. 102067, 2020.
76. F. Camilo, A. Meneely, and M. Nagappan, "Do bugs foreshadow vulnerabilities? a study of the chromium project," in *12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 269–279.
77. C. Joshi, U. K. Singh, and K. Tarey, "A review on taxonomies of attacks and vulnerability in computer and network system," *International Journal*, vol. 5, no. 1, 2015.
78. F. Mercaldo, A. Di Sorbo, C. A. Visaggio, A. Cimitile, and F. Martinelli, "An exploratory study on the evolution of android malware quality," *Journal of Software: Evolution and Process*, vol. 30, no. 11, p. e1978, 2018.

79. P. J. Morrison, R. Pandita, X. Xiao, R. Chillarege, and L. Williams, "Are vulnerabilities discovered and resolved like other defects?" *Empirical Software Engineering*, vol. 23, no. 3, pp. 1383–1421, 2018.
80. E. R. Russo, A. Di Sorbo, C. A. Visaggio, and G. Canfora, "Summarizing vulnerabilities' descriptions to support experts during vulnerability assessment activities," *Journal of Systems and Software*, vol. 156, pp. 84–99, 2019.
81. T. Mariani and S. R. Vergilio, "A systematic review on search-based refactoring," *Information and Software Technology*, vol. 83, pp. 14–34, 2017.
82. M. O'Keeffe and M. O. Cinnéide, "Search-based refactoring for software maintenance," *Journal of Systems and Software*, vol. 81, no. 4, pp. 502–516, 2008.
83. V. Alizadeh, M. A. Ouali, M. Kessentini, and M. Chater, "Refbot: Intelligent software refactoring bot," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 823–834.
84. L. Erlenhov, F. G. de Oliveira Neto, R. Scandariato, and P. Leitner, "Current and future bots in software development," in *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*. IEEE, 2019, pp. 7–11.
85. C. Lebeuf, M.-A. Storey, and A. Zagalsky, "Software bots," *IEEE Software*, vol. 35, no. 1, pp. 18–23, 2017.
86. I. Beschastnikh, M. F. Lungu, and Y. Zhuang, "Accelerating software engineering research adoption with analysis bots," in *2017 IEEE/ACM 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER)*. IEEE, 2017, pp. 35–38.
87. W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *International Symposium On Software Reliability Engineering*. IEEE, 1997, pp. 264–274.
88. G. Soares, R. Gheyi, D. Serey, and T. Massoni, "Making program refactoring safer," *IEEE software*, vol. 27, no. 4, pp. 52–57, 2010.
89. B. v. Bladel and S. Demeyer, "Test behaviour detection as a test refactoring safety," in *International Workshop on Refactoring*, 2018, pp. 22–25.
90. G. Soares, R. Gheyi, and T. Massoni, "Automated behavioral testing of refactoring engines," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 147–162, 2012.
91. F. Palomba, D. A. Tamburri, F. Arcelli Fontana, R. Oliveto, A. Zaidman, and A. Serebrenik, "Beyond technical aspects: How do community smells influence the intensity of code smells?" *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.
92. I. Kwan, A. Schroter, and D. Damian, "Does socio-technical congruence have an effect on software build success? a study of coordination in a software project," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 307–324, 2011.

93. G. Rodríguez-Pérez, G. Robles, and J. González-Barahona, "Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm," *Information and Software Technology*, vol. 99, pp. 164–176, 2018.
94. G. Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, and R. Oliveto, "Evaluating szz implementations through a developer-informed oracle," in *International Conference on Software Engineering (ICSE)*. IEEE, 2021, p. to appear.
95. N. Yoshida, T. Saika, E. Choi, A. Ouni, and K. Inoue, "Revisiting the relationship between code smells and refactoring," in *International Conference on Program Comprehension (ICPC)*. IEEE, 2016, pp. 1–4.
96. D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. de Mello, B. Fonseca, M. Ribeiro, and A. Chávez, "Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects," in *Joint Meeting on Foundations of Software Engineering*, 2017, pp. 465–475.
97. M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063–1088, 2017.
98. M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.
99. H. Mumtaz, M. Alshayeb, S. Mahmood, and M. Niazi, "An empirical study to improve software security through the application of code refactoring," *Information and Software Technology*, vol. 96, pp. 112–125, 2018.
100. S. Ghaith and M. Ó. Cinnéide, "Improving software security using search-based refactoring," in *International Symposium on Search Based Software Engineering*. Springer, 2012, pp. 121–135.
101. B. Alshammari, C. Fidge, and D. Corney, "Assessing the impact of refactoring on security-critical object-oriented designs," in *Asia Pacific Software Engineering Conference*. IEEE, 2010, pp. 186–195.
102. K. Maruyama and T. Omori, "A security-aware refactoring tool for java programs," in *Workshop on Refactoring Tools*, 2011, pp. 22–28.
103. B. Pérez, C. Castellanos, D. Correal, N. Rios, S. Freire, R. Spínola, and C. Seaman, "What are the practices used by software practitioners on technical debt payment: results from an international family of surveys," in *International Conference on Technical Debt*, 2020, pp. 103–112.
104. Z. Codabux and B. Williams, "Managing technical debt: An industrial case study," in *International Workshop on Managing Technical Debt (MTD)*. IEEE, 2013, pp. 8–15.
105. Z. Codabux, B. J. Williams, and N. Niu, "A quality assurance approach to technical debt," in *International Conference on Software Engineering Research and Practice (SERP)*, 2014.
106. E. Zabardast, J. Gonzalez-Huerta, and D. Šmite, "Refactoring, bug fixing, and new development effect on technical debt: An industrial case

study," in *Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2020, pp. 376–384.
107. M. Mohan, D. Greer, and P. McMullan, "Technical debt reduction using search based automated refactoring," *Journal of Systems and Software*, vol. 120, pp. 183–194, 2016.
108. M. Ghafari, P. Gadient, and O. Nierstrasz, "Security smells in android," in *the 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sep. 2017, pp. 121–130.
109. A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 164–175.