How Does Refactoring Impact Security When Improving Quality? A Security Aware Refactoring

Chaima Abid, Marouane Kessentini, Vahid Alizadeh, Mouna Dhaouadi and Rick Kazman

Abstract—While state of the art of software refactoring research uses various quality attributes to identify refactoring opportunities and evaluate refactoring recommendations, the impact of refactoring on the security of software systems when improving other quality objectives is under-explored. It is critical to understand how a system is resistant to security risks after refactoring to improve quality metrics. For instance, refactoring is widely used to improve the reusability of code, however such an improvement may increase the attack surface due to the created abstractions. Increasing the spread of security-critical classes in the design to improve modularity may result in reducing the resilience of software systems to attacks. In this paper, we investigated the possible impact of improving different quality attributes (e.g. reusability, extendibility, etc.), from the QMOOD model, effectiveness on a set of 8 security metrics defined in the literature related to the data access. We also studied the impact of different refactorings on these static security metrics. Then, we proposed a multi-objective refactoring recommendation approach to find a balance between quality attributes and security based on the correlation results to guide the search. We evaluated our tool on 30 open source projects. We also collected the practitioner perceptions on the refactorings recommended by our tool in terms of the possible impact on both security and other quality attributes. Our results confirm that developers need to make trade-offs between security and other qualities when refactoring software systems due to the negative correlations between them.

Index Terms—Quality, critical code, security metrics, attack surface, refactoring, multi-objective search.

1 Introduction

The National Institute of Standards and Technology (NIST) estimates that the US economy loses an average of $60 billion per year by either implementing patches to fix security vulnerabilities or the impact of these security issues [1], [2]. These vulnerabilities depend on how a system is designed and implemented. At the same time, code quality is also critical: it impacts programmer productivity and may cause project failure as maintenance consumes over 70% of the lifetime budget of a typical software project.

The ISO/IEC-25000 *SQuaRE* (Software product Quality Requirements and Evaluation) [3] classifies software quality in a structured set of eight characteristics and sub-characteristics. In this classification, security is a new characteristic that was created to measure how much a software is resistant to attacks and risks. Therefore, it is crucial to take this characteristic into account when improving the quality of the software.

Several researchers and practitioners have assumed that improving a quality metric of software, such as modularity, will have a positive impact on security, making the design more robust and resilient to attacks [4], [5], [6] . However, this assumption is poorly supported by empirical validations. Architects and developers may not pay much attention to design fragments containing data and logic pertinent to security properties, which makes them overexposed while still improving some quality aspects of their architecture. For instance, a developer may create a hierarchy in a set of classes to improve the reusability of the code. However, these actions may expand the attack surface if the superclass contains critical attributes and methods. Another example that we observed in practice is that improving modularity may result in spreading dependencies on security-critical files into many other components. A security-critical file contains data (e.g., attributes) and logic (e.g., methods) that can potentially be misused to violate fundamental security properties such as confidentiality, integrity, or availability of a system.

Refactoring to improve the design structure while preserving behavior is widely used to enhance the quality of software systems [7]. Most existing refactoring research focuses on handling conflicting quality attributes [8], [9], [10], [11], [12]. However, the impact of refactoring on security is poorly understood and under-studied. Recent studies estimate the impact of a few refactoring operations on some security metrics based on their definitions, but without empirically validating these assumptions on real software projects [13], [14], [15], [16]. To the best of our knowledge, there is no previous research on the correlations between security metrics and quality attributes, or that provided a tool to recommend refactorings based on the preferences of developers from both quality and security perspectives, and the possible conflicts between them.

In this paper, we investigate the possible correlations between the Quality Model for Object-Oriented Design (QMOOD) quality attributes [17] and a set of security metrics extracted from source code widely used in the current literature and practice [18], [19]. We also empirically validated the impact of different refactoring types on 8 code

- *Chaima Abid, Marouane Kessentini, Vahid Alizadeh, Mouna Dhaouadi are with the department of Computer and Information Science, University of Michigan, Dearborn, MI, USA.*
  E-mail: firstname@umich.edu
- *Rick Kazman is a Professor at the University of Hawaii and a Principal Researcher at the Software Engineering Institute of Carnegie Mellon University. E-mail: kazman@hawaii.edu*

Manuscript received on November 2019.

security metrics that are primarily related to data access.

We analyzed a total of 30 open-source projects and, based on the outcomes of these analyses showing the conflicting nature of the studies security and quality metrics, we propose a security-aware multi-objective refactoring approach to find a balance between code qualities and security metrics. We formulated the different quality and security objectives as fitness functions to guide the search for relevant refactorings and find trade-offs between them using NSGA-II [20].

We evaluated our tool on this set of 30 projects. Furthermore, we compared our results with an existing multi-objective refactoring tool [9] that only considers code quality, to understand the sacrifice in security measures when improving code quality and vice-versa. The comparison shows that our security-aware approach performed better than the existing approach when it comes to improving the security of systems, and with low cost in terms of sacrificing code quality. Our survey of 15 practitioners confirmed the efficiency of our tool and the importance of considering security while improving other qualities. More details about the surveys, experiments and tool can be found in the online appendix [21].

The primary contributions of this paper are as follows:

1) The paper introduces one of the first empirical studies to understand the impact of source code refactoring on both quality and security metrics and the correlations between them.
2) The creation of a framework to recommend refactorings to find trade-offs between quality and security objectives considering the correlation results between them.
3) A validation of this framework on open source systems. The survey with practitioners shows the potential of our work in improving refactoring recommendations by taking into account both security and quality.

The remainder of this paper is organized as follows: Section 2 introduces the background and motivations behind our work, Section 3 presents the description of our security-aware multi-objective approach while Section 4 contains the results of our methodology. Section 5 discusses threats to validity. Section 6 surveys relevant related work, and finally, we conclude and outline our future research directions in Section 7.

2 Background and Motivating Example

In this section, we present first the necessary background related to quality attributes, security metrics, and refactoring operations. Then, we describe a motivating example related to the possible negative impact of refactoring on security.

2.1 Background

2.1.1 Quality Attributes

We selected as code quality metrics the ones defined by ISO 9126, called QMOOD [22], since they are commonly used in industry to estimate code quality [23], [24], [25], cover most of the maintainability issues, and are also frequently used in refactoring studies [9], [11], [12], [26], [27], [28]. The QMOOD model contains six quality attributes—reusability, flexibility, understandability, functionality, extendibility, and effectiveness as described in Table 1.

2.1.2 Security Metrics

Code elements containing confidential or sensitive information such as userIDs, transactions, credit card, authentication, security constraints, may be security-critical. These code elements may be attributes, methods, classes, or packages. If these code fragments are over-exposed, this may result in vulnerabilities that can be exploited. Thus, developers should ensure that these code fragments are not over-exposed. Several software security metrics have been defined in the research literature at different levels of abstraction [29]. We focus in this study on those that are related to the code level and can be measured by static and dynamic analyses.

For the selected security metrics, we have adopted the terminology and definitions proposed in existing studies [18], [19]. We consider that classified, confidential, and vulnerable attributes all refer to attributes that need to be secured. Tables 2 and 3 summarizes the definition of these 8 security metrics: Classified Instance Data Accessibility (CIDA), Classified Class Data Accessibility (CCDA), Classified Operation Accessibility (COA), Classified Mutator Attribute Interactions (CMAI), Classified Accessor Attribute Interactions (CAAI), Classified Attributes Interaction Weight (CAIW), Classified Methods Weight (CMW) and Vulnerable Association within a class (VAClass). We adopted the Soot parser [30], based on static analysis, to calculated these metrics including the automated identification of classified versus non-classified code elements, as shown in the video of our tool [21].

We chose only those 8 security metrics among the ones available in the literature specifically because their definition is clear and relatively easy to implement. We also wanted to highlight that our parser is based on Soot [30] for static analysis–we did not create a custom parser from scratch. The source code is analyzed to extract the relevant code elements such as classes, methods, attributes, etc. and the relationships between them. Each code element has several attributes that describe its level of access/visibility, whether it is considered to be classified or not.

We describe in the following the different steps to identify the security sensitive attributes by taking inspiration from existing studies [31], [32], [33], [34] based on text similarities/mining. We first use a set of keywords [31], [33], [34] related to security and indicators of sensitive information extracted from multiple sources such as source codes, comments, security bugs, vulnerability reports, commit messages, and security questions/tags on Stack Overflow. We included these keywords in the online appendix associated with this submission. Second, we calculated a textual criticality score, based on cosine similarity, for each file to estimate the extent to which the file is related to security concerns. The higher the score is the more likely the file needs to be protected. We pre-processed the source code using tokenization, lemmatization, stop words filtering and punctuation removal [31], [33]. Then, we computed the cosine similarity between each file and the set of keywords. Finally, we manually validated the top 10 critical files and use their critical attributes (fields that have names that match one of the keywords from the list we gathered at the beginning) to identify the critical attributes

TABLE 1: QMOOD quality attributes

Metric	Definition	Formula
Reusability	Reflects the presence of object-oriented design characteristics that allow a design to be reapplied to a new problem without significant effort.	-0.25*Coupling + 0.25*Cohesion + 0.5*Messaging + 0.5*Design Size
Flexibility	Characteristics that allow the incorporation of changes in a design. The ability of a design to be adapted to provide functionality related capabilities.	0.25*Encapsulation - 0.25*Coupling + 0.5*Composition + 0.5*Polymorphism
Understandability	The properties of designs that enable it to be easily learned and comprehended. This directly relates to the complexity of the design structure.	-0.33*Abstraction + 0.33*Encapsulation + 0.33*Coupling + 0.33*Cohesion - 0.33*Polymorphism - 0.33*Complexity - 0.33* Design Size
Functionality	The responsibility assigned to the classes of a design, which are made available by classes through their public interfaces.	0.12*Cohesion + 0.22*Polymorphism + 0.22*Messaging + 0.22*Design Size + 0.22*Hierarchies
Extendibility	Refers to their presence and usage of properties in an existing design that allow for the incorporation of new requirements in the design.	0.5*Abstraction - 0.5*Coupling + 0.5*Inheritance + 0.5* Polymorphism
Effectiveness	This refers to the designs ability to achieve the desired functionality and behavior using object oriented design concepts and techniques.	0.2*Abstraction + 0.2*Encapsulation + 0.2*Composition + 0.2*Inheritance + 0.2*Polymorphism

TABLE 2: Security metrics terminology.

Term	Definition
Classified Attribute	An attribute which is defined in UMLsec [35] as secrecy.
Instance Attribute	An attribute which value is stored by each instance of a class.
Class Attribute	An attribute which value is shared by all instances of that class.
Classified Methods	A method which interacts with at least one classified attribute.
Mutator	A method that sets the value of an attribute.
Accessor	A method that returns the value of an attribute.

in all the other files that will be used to compute the security metrics. This process, including security metrics calculation, is not time-consuming since it takes a few seconds to minutes to extract the security-critical attributes and compute the metrics, depending on the size of the project to be analyzed. We note that the identification of code elements as security sensitive is not a core contribution of this paper since we leveraged the use of existing work for this step.

2.1.3 Refactoring

Martin Fowler defined refactoring as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior" [7]. This implies that refactoring is a method that reconfigures code structure, without altering its behavior, to improve code quality in terms of maintainability, extensibility, and reusability. Table 4 summarizes 15 refactoring types considered in this paper. Recent empirical studies on refactoring show that these refactorings are widely used in open-source projects [15], [27], [36].

2.2 Motivating Examples

By mining the well-known Common Vulnerabilities and Exposures (CVE) security bug database, we found a total of 269 security vulnerabilities that were introduced by code refactorings. These 269 vulnerabilities were manually identified by the authors of this paper out of 681 reports containing the keyword "refactor". Figure 1 shows an example of a vulnerability, **CVE-2019-13177**[1], from Django REST Registration library due to refactorings resulted in allowing remote attackers to trick the verification process. This security bug impacted the confidentially of Django REST Registration library in several releases before 0.5.0. Thus, it is essential to evaluate the impact of the recommended refactorings on the security of the application.

We introduce, in the following, another motivating example to show how refactoring may improve code quality while making the design weaker from a security perspective. The design fragment in Figure 2 is responsible for storing information about customer accounts, which, by definition, requires careful attention in terms of security to access those classes. A bank account can be either a debit or credit account. The *interestRateConstant* is an attribute that stores the value of the interest rate of the credit account. Thus, it is only used by the *creditAccount* class. The deposit and withdraw operations have duplicated code that performs the transactions. This code can be extracted to a new separate method that can be used by both operations. Both *accountNumber* and *creditCardNum* are sensitive and are meant to be kept confidential.

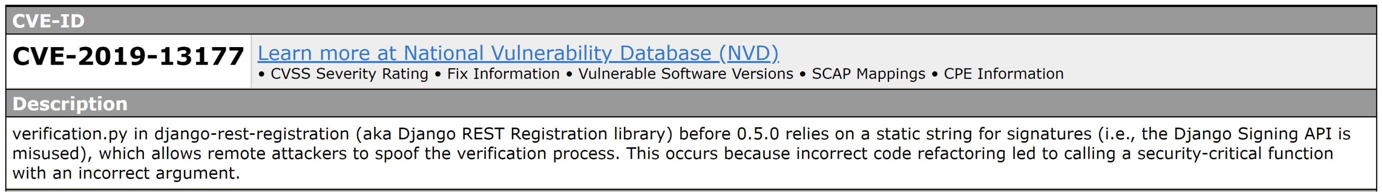
The developer applied the refactoring "push down field" by moving the *interestRateConstant* from the *BankAccount* class to its subclass *CreditAccount* as well as the refactoring "extract method" by moving the duplicated code to a separate new method called *performTransaction* and replacing the old code with a call to this new method. These refactorings improved cohesion and messaging [27], which results in increasing the following quality attributes: Understandability, Functionality, and Reusability [37]. However, these transformations might increase the security metrics CMAI, CAAI, CMW, and CAIW [15] which will reduce the security of the design due to the fact that the classes are becoming more exposed and easier to access than before. This example motivates our research to investigate further the impact of refactorings on security when improving code quality.

3 Security-Aware Multi-Objective Refactoring

3.1 Overview

Our approach, as sketched in figure 3, takes as input the source code (or GitHub link) of a project to be analyzed and generates a list of refactoring recommendations that balance code quality and security based on developer preferences.

1. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-13177



CVE-ID	
CVE-2019-13177	Learn more at National Vulnerability Database (NVD) • CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information
Description	
verification.py in django-rest-registration (aka Django REST Registration library) before 0.5.0 relies on a static string for signatures (i.e., the Django Signing API is misused), which allows remote attackers to spoof the verification process. This occurs because incorrect code refactoring led to calling a security-critical function with an incorrect argument.	

Fig. 1: An example of a security vulnerability from Django REST Registration library due to refactorings.

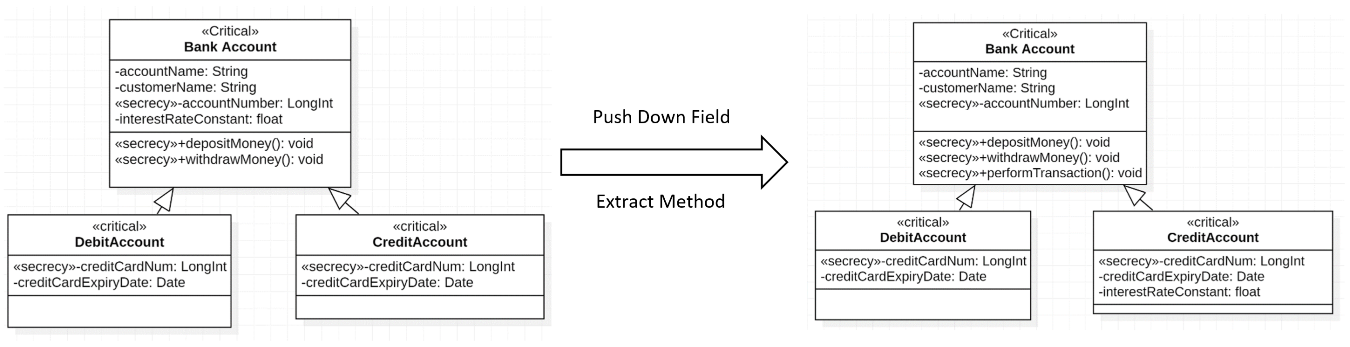

«Critical»
Bank Account
-accountName: String
-customerName: String
«secrecy»-accountNumber: LongInt
-interestRateConstant: float
«secrecy»+depositMoney(): void
«secrecy»+withdrawMoney(): void
«critical»
DebitAccount
«secrecy»-creditCardNum: LongInt
-creditCardExpiryDate: Date
«critical»
CreditAccount
«secrecy»-creditCardNum: LongInt
-creditCardExpiryDate: Date
Push Down Field
Extract Method
«Critical»
Bank Account
-accountName: String
-customerName: String
«secrecy»-accountNumber: LongInt
«secrecy»+depositMoney(): void
«secrecy»+withdrawMoney(): void
«secrecy»+performTransaction(): void
«critical»
DebitAccount
«secrecy»-creditCardNum: LongInt
-creditCardExpiryDate: Date
«critical»
CreditAccount
«secrecy»-creditCardNum: LongInt
-creditCardExpiryDate: Date
-interestRateConstant: float

Fig. 2: A Simplified Bank Account System Hierarchy Before and After Refactoring

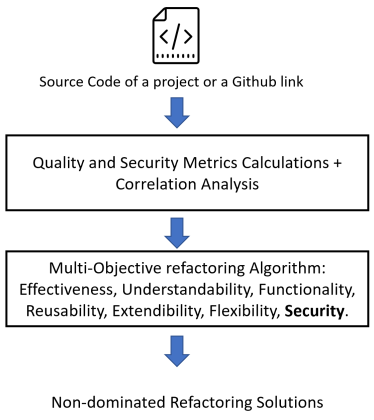

Source Code of a project or a Github link
Quality and Security Metrics Calculations + Correlation Analysis
Multi-Objective refactoring Algorithm: Effectiveness, Understandability, Functionality, Reusability, Extendibility, Flexibility, Security.
Non-dominated Refactoring Solutions

Fig. 3: Security-Aware Multi-Objective Refactorings

The first component parses the code to calculate the security metrics and quality attributes as defined in the previous section. Then, the collected data is used to analyze the correlation between the different quality and security metrics (without the need for refactoring at this point).

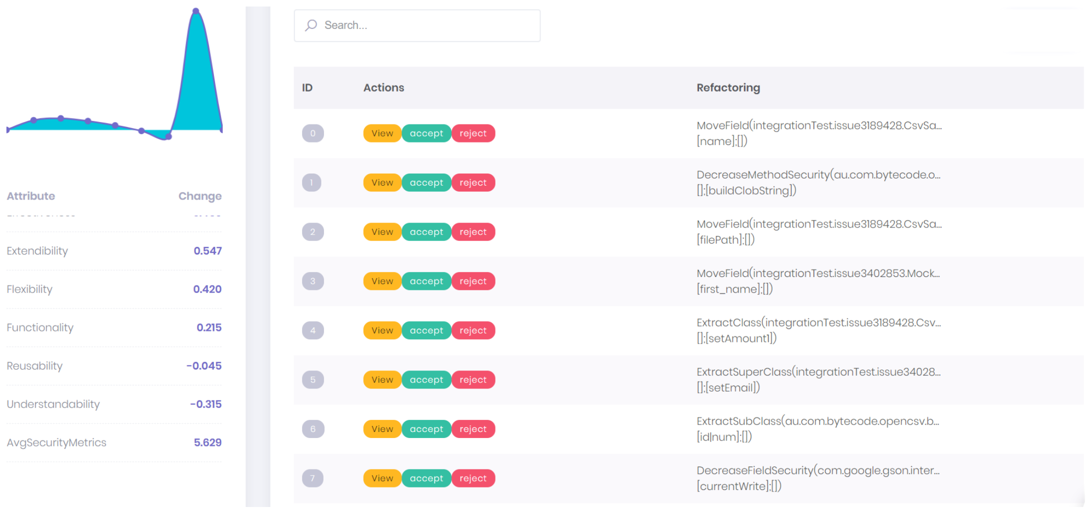
For the second component, we adapted a multi-objective search algorithm, based on NSGA-II [20], used in our previous work [9] to integrate the security and quality objectives. We selected this algorithm due to its ability to find trade-offs between independent or conflicting objectives, and it has previously been applied for various software engineering problems [8], [9], [10], [11], [12]. Security and quality objectives cannot be aggregated together since they are independent and even conflicting, as discussed later in our validation. Our goal is to find a set of non-dominated refactoring solutions capable of improving both the quality and security of the project taken as input. A code refactoring activity may be focused on quality improvements, and the developers care less about security (e.g., the component is used internally and never exposed to attacks). In this case, users of the tool may want to assign higher weights to quality metrics. In another scenario, it could be the opposite, especially for critical code fragments. In our multi-objective formulation, the developer is not required to enter any weights to the objectives since the output of the algorithm is a Pareto-front of a diverse set of solutions that the user can select one of them based on their preferences. Finally, a user can interact with our tool to accept or reject the refactoring recommendations. A detailed demo can be found in [21]. In the remainder of this section, we will explain the steps of the approach.

3.2 Algorithm Adaptation

The search space is composed of the different refactoring operations as well as an exhaustive combination of code locations, attributes, and methods. The algorithm is executed for some iterations to find non-dominated solutions balancing the 7 objectives of improving the 6 QMOOD quality metrics, and the last objective of minimizing the security objective (aggregating the 8 security metrics) in the proposed solutions. The output of this step is a set of Pareto-equivalent refactoring solutions that optimize the above objectives. These solutions are not dominated with respect to each other. In the following subsections, we summarize the adaptation of the multi-objective algorithm to our problem.

3.2.1 Solution Representation.

In our approach, a refactoring solution is represented by an ordered vector of refactoring operations as shown in Figure 4. Each operation is defined by an action (e.g., pull up field, encapsulate field, extract superclass, etc.) and its parameters such as source class, target class, attributes, etc. Since we need to apply the change operators of mutation and crossover during evolution, we need to evaluate the feasibility of a solution and see if it preserves the behavior of



Search...

Attribute	Change
Extendibility	0.547
Flexibility	0.420
Functionality	0.215
Reusability	-0.045
Understandability	-0.315
AvgSecurityMetrics	5.629

ID	Actions	Refactoring
0	View accept reject	MoveField(integrationTest.issue3189428.CsvSa... [name];[])
1	View accept reject	DecreaseMethodSecurity(au.com.bytecode.o... [];[buildClobString])
2	View accept reject	MoveField(integrationTest.issue3189428.CsvSa... [filePath];[])
3	View accept reject	MoveField(integrationTest.issue3402853.Mock... [first_name];[])
4	View accept reject	ExtractClass(integrationTest.issue3189428.Csv... [];[setAmount1])
5	View accept reject	ExtractSuperClass(integrationTest.issue34028... [];[setEmail])
6	View accept reject	ExtractSubClass(au.com.bytecode.opencsv.b... [idlnum];[])
7	View accept reject	DecreaseFieldSecurity(com.google.gson.inter... [currentWrite];[])

Fig. 4: Sample of outputs (refactorings) of our Web app on the Open CSV project to balance quality and security.

the system using a set of pre- and post-conditions defined in [38].

3.2.2 *Fitness Functions.*

Our approach takes into consideration seven objectives: the first six are the relative changes of the 6 QMOOD attributes [17] after applying a refactoring solution. QMOOD defines six high-level design quality attributes (reusability, flexibility, understandability, functionality, extendibility, and effectiveness) that are calculated using 11 lower-level metrics [9]. Each objective can be written as follow:

$$QualityObjective_i = \frac{Q_i^{after} - Q_i^{before}}{Q_i^{before}} \quad (1)$$

where Q_i^{before} and Q_i^{after} are the values of the $qualityAttribute_i$ before and after applying a refactoring solution, respectively.

Since all metrics in the table 3 are at the class level, we consider the corresponding system-level metrics as the average of all class level metrics. For instance, *AvgCCDA* is defined as the ratio of the sum of the *CCDA* values of all classes of the system to the number of classes in that system. In a similar manner, we define the other system-level security metrics *AvgCIDA*, *AvgCOA*, *AvgCAAI*, *AvgCMAI*, *AvgCMW*, *AvgCAIW* and *AvgVA*. Therefore, the seventh objective, which is the security objective, corresponds to the relative change in the average of the average of all eight security metrics in the table 3 after applying a refactoring solution. We can represent the fitness function of the security objective as follows:

$$SecurityObjective = \frac{S^{after} - S^{before}}{S^{before}} \quad (2)$$

where S= (*AvgCCDA* + *AvgCIDA* + *AvgCOA* + *AvgCAAI* + *AvgCMAI* + *AvgCMW* + *AvgCAIW* + *AvgVA*) / 8

Unlike the quality objectives, we decided to aggregate the security metrics into one objective since they are not conflicting to each other based on our analysis of the data on the open-source systems detailed later in our experiments. Furthermore, the performance of the multi-objective algorithm will decrease when the number of objectives becomes large.

4 Experiments and Results

We used a set of 30 open source projects to study the possible correlations between 1) the quality and security metrics and 2) refactoring types and security metrics. To evaluate the ability of our security-aware multi-objective refactoring tool to generate good refactoring recommendations that balance both quality and security, we conducted a set of experiments based on 4 out of the 30 open source systems. The obtained results are subsequently statistically analyzed with the aim of comparing our proposal with a variety of existing approaches. The relevant data related to our experiments and a demo about the main features of the tool can be found in [21]. We have also conducted a survey with practitioners to manually evaluate the refactoring recommendations and the obtained correlations between quality, refactoring types and security.

In this section, we first present our research questions and validation methodology followed by the experimental setup. Then we describe and discuss the obtained results.

4.1 Research Questions

In this study, we defined four main research questions:

- **RQ1: Impact of refactoring on code security.** Can automated refactoring have a significant impact on security metrics?
- **RQ2: Impact of improving quality on security and vice-versa.** Are there strong correlations between code quality attributes, as measured by the QMOOD metrics, and code security metrics?
- **RQ3: Comparison with an existing work for refactoring recommendation** How does our security-aware refactoring tool perform compared to refactoring approaches that only focus on improving quality (and not security)?
- **RQ4: Insights.** Do professional programmers highly value considering security while improving quality?

To answer RQ1, we collected a dataset of refactorings applied on 30 medium to large-size open-source systems, listed in table 5, to understand the impact of 14 refactoring

TABLE 3: Security metrics definition

Metric	Definition
Classified Instance Data Accessibility (CIDA)	consider CA as a set of classified attributes in a class C, $CA = ca_i, i \in \{1,2,\ldots,n\}$, and CIPA its classified public attributes as $CIPA = cipa_i, i \in \{1,2,\ldots,n\}$ $CIDA(C) = \|CIPA\|/\|CA\|$
Classified Class Data Accessibility (CCDA)	consider CA as a set of classified attributes in a class C, $CA = ca_i, i \in \{1,2,\ldots,n\}$, and CCPA its classified class public attributes as $CCPA = ccpa_i, i \in \{1,2,\ldots,n,\}$ $CCDA(C) = \|CCPA\|/\|CA\|$
Classified Operation Accessibility (COA)	consider CM as a set of classified methods in a class C, $CM = cm_i, i \in \{1,2,\ldots,n\}$, and CPM classified public methods as $CPM = cpm_i, i \in \{1,2,\ldots,n\}$ $COA(C) = \|CPM\|/\|CM\|$
Classified Mutator Attribute Interactions (CMAI)	consider a set of mutator methods in a class C as $MM = mm_i, i \in \{1,2,\ldots,mm\}$, and CA the classified attributes $CA = ca_j, j \in \{1,2,\ldots,ca\}$. Let $\alpha(\mathrm{CA}_j)$ be the number of mutator methods which may access classified attribute (CA_j). Then, CMAI can be expressed as: $CMAI(C) = \sum_{j=1}^{ca} \alpha(CA_j)/(\|MM\| * \|CA\|)$
Classified Accessor Attribute Interactions (CAAI)	consider a set of accessor methods in a class C as $AM = mm_i, i \in \{1,2,\ldots,am\}$, and CA the classified attributes $CA = ca_j, j \in \{1,2,\ldots,ca\}$. Let $\beta(\mathrm{CA}_j)$ be the number of accessor methods which may access classified attribute (CA_j). Then, CAAI can be expressed as: $CAAI(C) = \sum_{j=1}^{ca} \beta(CA_j)/(\|AM\| * \|CA\|)$
Classified Attributes Interaction Weight (CAIW)	consider a set of classified attributes CA in a class C as $CA = ca_i, i \in \{1,2,\ldots,ca\}$, and A the set of attributes $A = a_j, j \in \{1,2,\ldots,a\}$. Let ($CA_j$) be the number of methods which may access classified attribute (CA_j), and $\theta(\mathrm{A}_i)$ be the number of methods which may access the attribute (A_i), Then, CAIW can be expressed as: $CAIW(C) = \sum_{j=1}^{ca} \gamma(CA_j)/\sum_{i=1}^{a} \theta(A_i)$
Classified Methods Weight (CMW)	consider CM as a set of classified methods in a class C, $CM = cm_i, i \in \{1,2,\ldots,m\}$, and M the set of all methods as $M = m_j, j \in \{1,2,\ldots,n\}$ $COA(C) = \|CM\|/\|M\|$
Vulnerable Association with in a class (VAClass)	consider CA as a set of classified attributes in a class C, $CA = ca_i, i \in \{1,2,\ldots,m\}$, and M the set of all methods as $M = m_j, j \in \{1,2,\ldots,n\}$, and $\alpha(\mathrm{M}_j)$ the number of classified attributes associated with the method m_j. Then VAClass is: $VAClass(C) = \sum_{j=1}^{n} \alpha(m_j)/(\|CA\| * \|M\|)$

TABLE 4: Refactoring Types Considered in our Study

Refactoring Types	Definition
Encapsulate Field	Changes the access modifier of public fields to private and generates its getter and setter.
Increase Field Security	Changes the access modifier of protected fields to private, and of public fields to protected.
Decrease Field Security	Changes the access modifier of protected fields to public, and of private fields to protected.
Pull Up Field	If two subclasses have the same field, then this rule moves this field to their superclass.
Push Down Field	If only some subclasses use a field, then this rule moves this field to those subclasses.
Move Field	Moves a field to another class.
Increase Method Security	Changes the access modifier of protected methods to private, and of public methods to protected.
Decrease Method Security	Changes the access modifier of protected methods to public, and of private methods to protected.
Pull Up Method	If two subclasses have the same method, then this rule moves the method to their superclass.
Push Down Method	If only some subclasses use a method, then this rule moves the method to those subclasses.
Move Method	Moves a method to another class.
Extract Class	Creates a new class from an existing one.
Extract Superclass	If two subclasses have similar features, this rule creates a superclass and moves these features into it.
Extract Subclass	If two superclasses have similar features, this rule creates a subclass and moves these features into it.
Extract Method	takes a sequence of statements, copies them into a new method, and then replaces the original statements with an invocation of the new method.

types on 8 different security metrics. We selected these systems based on their domains, size and large history of evolution (e.g. commits). We did not extract refactorings from previous commits due to the challenges related to differentiating between functional and non-functional changes and the limited number of refactorings that developers apply manually. Instead, we obtained the data by running the refactoring recommendation tool of Alizadeh et al. [9] on these projects, selecting the obtained refactoring solution and recording its impact on the security metrics. We selected the tool based on its high accuracy in recommending relevant refactorings that significantly improve the quality. Then, we statistically analyzed the impact of these refactoring types on code security metrics for the 30 projects.

To answer RQ2, we used a procedure similar to the one used for RQ1: we collected data from the execution of our tool on the 30 projects by recording the impact of the refactorings on both the QMOOD quality attributes and the 8 code security metrics. Unlike the impact of QMOOD on quality, we note that the security level increases when the security metrics decrease. Finally, we ran statistical tests to understand the correlations between the different metrics using the Pearson correlation coefficient [39] (chosen due to the normal distribution of the data).

To answer RQ3, we compared our approach with an existing technique that considers only the QMOOD attributes [9] as objectives using 4 projects, as described later. Since meta-heuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. For this reason, our study is based on 30 independent simulation runs for each problem instance to make sure that the results were statistically significant. The goal of this research question is to understand the cost of improving code quality on security and vice-versa. We selected the work of Alizadeh et al. [9] since it is the closest to our proposed approach and outperformed most of the existing refactoring tools based on the same systems used in this evaluation. We only considered five systems in this comparison due to the very time-consuming task to run the different heuristic algorithms 30 times to check if the results are statistically significant. Furthermore, it is difficult to find knowledgeable participants who can manually evaluate the results on all 30 open source projects. Thus this part of our evaluation poses a threat to validity.

To answer RQ4, we used a post-study questionnaire to collect the opinions of developers regarding our tool and the relevance of considering security when refactoring. Furthermore, the participants manually evaluated the refactoring recommendations of our approach. We asked the developers about their opinions on the possible correlations between 1) quality and security metrics; and 2) refactoring types and security. The survey allowed us to compare the quantitative results obtained in our experiments with developer opinions. The full details of our extensive validation, including a demo of our tool and the survey details, can be found at [21].

4.2 Software Projects and Experimental Setting

4.2.1 Studied Projects

We used a set of 30 well-known open-source Java projects as detailed in Table 5. We selected these systems for our validation because they range from medium to large-sized and have been actively developed in recent years. Table 5 also provides some descriptive statistics about these programs.

TABLE 5: Studied Open Source Projects.

System	Release	#Classes	KLOC	GitHub Link
jFreeChart	v1.0.9	521	170	jfree/jfreechart.git
ArgoUML	v0.3	1358	114	marcusvnac/argouml-spl.git
atomix	v3.0.11	2719	188	atomix/atomix.git
JHotDraw	v7.5.1	585	25	wumpz/jhotdraw.git
GanttProject	v1.10.2	241	48	bardsoftware/ganttproject.git
Apache Ant	v1.8.2	1191	112	apache/ant.git
moshi	v1.8.0	289	27	square/moshi.git
opencsv	v1.7	50	7	jlawrie/opencsv.git
zerocell	v0.3.2	39	3	creditdatamw/zerocell.git
gson	v2.8.5	691	69	google/gson.git
jolt	v0.1.1	370	31	bazaarvoice/jolt.git
Hystrix	v1.5.18	1117	85	Netflix/Hystrix.git
btm	v2.1.3	375	40	bitronix/btm.git
packr	v1.2	8	3	libgdx/packr.git
tracer	v2.0.0	33	3	zalando/tracer.git
JSAT	v0.0.9	1171	185	EdwardRaff/JSAT.git
smile	v1.5.2	1206	8316	haifengl/smile.git
dkpro-core	v1.10.0	1269	1323	dkpro/dkpro-core.git
Erdos	v1.0	128	7	Erdos-Graph-Framework/Erdos.git
jgrapht	v1.3.0	1257	171	jgrapht/jgrapht.git
mockito	v2.27.3	1880	94	mockito/mockito.git
tablesaw	v0.32.7	583	714	lwhite1/tablesaw.git
bazel	v0.25.0	11267	2753	bazelbuild/bazel.git
spotbugs	v4.0.0	5207	389	spotbugs/spotbugs.git
FreeBuilder	v2.3.0	1636	58	google/FreeBuilder.git
async-http	v2.8.1	602	52	AsyncHttpClient/async-http-client.git
javaparser	v3.13.10	1414	251	javaparser/javaparser.git
vavr	v0.10.0	838	135	vavr-io/vavr.git
javamelody	v1.77.0	662	109	javamelody/javamelody.git
commons-cli	v1.4	63	10	apache/commons-cli.git

4.2.2 Subjects

Our qualitative study involved 15 software developers. All participants were volunteers who were knowledgeable in software security, Java, refactoring, and quality assurance. They were all hired from our former and current industry partners of refactoring projects.

Participants were first asked to fill out a pre-study questionnaire. The questionnaire helped to collect background information such as their role within the company, their programming experience, and their familiarity with software security, quality, and refactoring. They all had a minimum of 2 years of experience as programmers and 5 out of 15 have over 5 years of experience. 12 participants were working on software assurance tasks as part of their regular duties, which was one of the main criteria used to solicit their participation, based on our previous collaborations and contacts. The other criteria were related to their level of expertise in refactoring and security, and also their familiarity with the five selected open source systems. The pre-study survey shows that the majority of the developers (11 out 15) have high experience and are knowledge about software refactoring and security. A minimum of 12 of 15 participants per system have medium or above expertise regarding the evaluated open source systems. The full details of our pre-study survey results can be found at [21].

Each participant was asked then to complete an evaluation form to evaluate 5 refactoring solutions that had different impacts on quality and security on 4 different systems: JHotDraw, Gantt, Apache Ant and JFreeChart. The participants were asked to evaluate the refactorings on all the systems; we did not divide them into groups. After that, each participant was given a post-study survey. This second survey was more general as it collected the practitioners' opinions on the relevance of the outcomes and their perception of the importance of considering security when refactoring their code.

4.2.3 Parameter tuning and statistical tests

Parameter setting significantly influences the performance of a search algorithm on a problem. For this reason, for each algorithm and for each system, we performed a set of experiments using several population sizes: 50, 100, 200, 300 and 500. The stopping criterion was set to 10,000 evaluations for all algorithms to ensure fairness of comparison. The other parameter values were fixed by trial and error and are as follows: crossover probability = 0.8 and mutation probability = 0.5 where the probability of gene modification is 0.3. We also limited the size of the refactoring solutions to no more than 30 operations.

To have significant results, for each pair (algorithm, system), we used one of the most efficient and popular approaches for parameter setting of evolutionary algorithms which is Design of Experiments (DoE) [40]. Each parameter was uniformly discretized in some intervals. Values from each interval have been tested for our application. Finally, we picked the best values for all parameters. Hence, a reasonable set of parameter values were applied.

The following statistical tests show that all the comparisons performed between our approach and existing ones are statistically significant based on all the metrics and the systems considered in our experiments. We used a 2-sample t-test with a 95% confidence level (α = 5%) to find out whether our sample results of different approaches are significantly different. We also calculated the Pearson coefficient to study the various correlations.

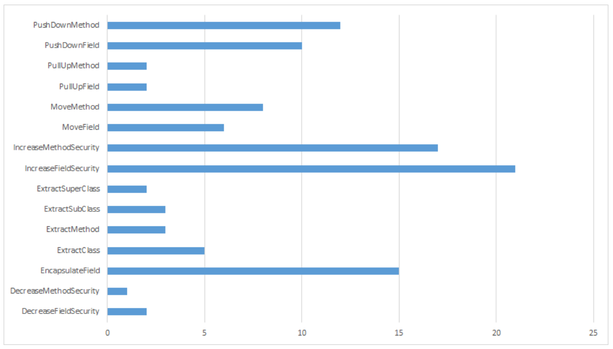


Fig. 5: Average distribution of the refactoring types among the solutions recommended for the 30 projects that significantly improve the security objective.

4.3 Results

Results for RQ1. Table 6 summarizes the correlations between the different types of refactorings and averaged security metrics considered in our experiments by analyzing the refactoring recommendations generated by our tool on the 30 projects. The results show either a positive or negative correlation based on the Pearson Correlation Coefficient except for the Move Field refactoring. The symbol "++" (strong positive correlation) means that the Pearson correlation coefficient has a value higher than 0.5 while "–" (Strong negative correlation) means the opposite (lower than -0.5). The symbol "+" means that the Pearson correlation coefficient is between 0.1 and 0.5 and "-" means the opposite (between -0.1 and -0.5). The symbol "*" reflects that the correlation coefficient is around 0 (between -0.1 and 0.1) and there is no statistically significant correlation. For each refactoring type, we filtered the solutions to keep only the ones containing that type and counted its occurrence within the solution. Then, we checked the correlation between the appearance of the refactoring type and its impact on the security metric.

Increase Field Security refactoring has the strongest positive correlation with the average of the 8 security metrics. It is expected that the frequent use of this refactoring type will reduce access to the attributes which may reduce their visibility and reduce the attack surface when a set of classes are exposed to malicious code. The same observation is also valid for Increase Method Security which also has a positive correlation with security improvements. Encapsulate Field, Push Down Field, and Push Down Method refactorings have also positive correlations with the security average measure. It is clear that all these refactoring types reduce the level of abstraction of classes which may increase the protection of the fields and methods.

Decrease Field Security, Decrease Method Security, and Extract Superclass have a strong negative correlation with the security measure since the Pearson Correlation Coefficient is lower than -0.6. All these refactorings can either make the fields and methods overexposed or increase the abstraction of the code which may have a negative impact on security. The Encapsulate Field refactoring increases the ability to conceal object data. Otherwise, all objects would be public and other objects could get and modify the object's data without any constraints. Furthermore, the encapsulate field refactoring can help in bringing data and behaviors closer together which will reduce unnecessary access and public visibility of attributes. Thus, the security metrics should be improved after application of Encapsulate Field refactorings.

Table 6 also shows that Extract Superclass is negatively correlated with the security metrics. One of the main explanations of this outcome is the fact that creating superclasses may expose all the child classes under the created superclass. Thus, the attack surface could be rapidly expanded when this refactoring type is extensively used. In fact, someone who has access to a superclass can affect its subclasses' behavior by modifying the implementation of an inherited method that is not overridden. If a subclass overrides all inherited methods, a superclass can still affect subclass behavior by introducing new methods.

Figure 5 and Table 7 show the most frequent refactorings and patterns in the solutions that significantly increased the security measure. In this study, a refactoring pattern is an ordered sequence of refactoring operations. We found that the most frequent refactoring types are the ones making the methods and fields less exposed and accessed, which confirms the correlation results. Figure 6 describes the impact of refactorings generated by our tool on the 8 security metrics aggregated into one objective. The results show that none of the metrics are conflicting with other security metrics since they were all minimized using the refactoring solutions. This observation confirms our choice to aggregate them rather than considering them as separate objectives. All the security metrics are normalized in the range of [0,1].

To summarize, refactoring can impact code security metrics both positively and negatively based on our analysis of the refactoring solutions proposed for 30 open source projects.

Finding 1: Encapsulate Field, Increase Field Security, Push Down Field, Increase Method Security, Push Down Method are all positively correlated with the avg security metrics. Decrease Field Security, Pull Up Field, Decrease Method Security, Pull Up Method, Move Method, Extract Class, Extract Superclass, Extract method and Extract Subclass are all negatively correlated with the avg security metrics. There is no statistically significant correlation between the Move Field refactoring and the avg security metrics.

TABLE 6: Correlation results between the average of security metrics and different refactoring types on the 30 projects. The results are statistically significant using the 2sample t-test with a 95% confidence level (α = 5%)

Refactoring \ Average Security Metrics	Pearson Correlation Coefficient
Encapsulate Field	+ (0.237)
Increase Field Security	++ (0.728)
Decrease Field Security	- - (-0.624)
Pull Up Field	- (0.361)
Push Down Field	+ (0.471)
Move Field	* (0.026)
Increase Method Security	+ (0.358)
Decrease Method Security	- - (-0.681)
Pull Up Method	- (-0.316)
Push Down Method	+ (0.247)
Move Method	- (-0.235)
Extract Class	- (-0.437)
Extract Superclass	- - (-0.694)
Extract Subclass	- (-0.424)
Extract method	- (- 0.472)

TABLE 7: The two most common refactoring patterns with the highest impact on the improvement of the average security measure for the 30 open source projects.

Refactoring patterns	Average Security Improvement
Encapsulate Field, Increase Field Security, Increase Method Security, Push Down Field, Move Method	0.42
Increase Field Security, Increase Method Security, Move Field, Push Down Method	0.34

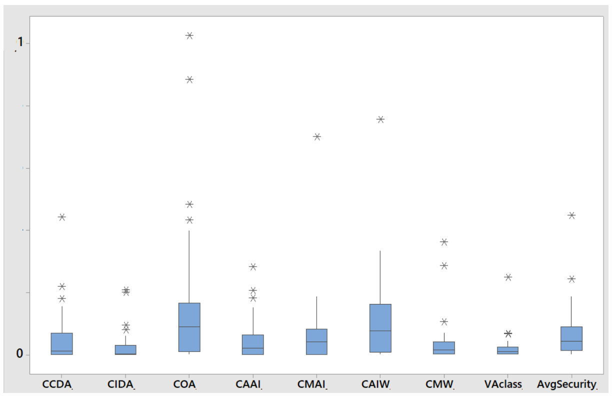


Fig. 6: Impact of the recommended refactorings on security metrics based on the 30 projects.

Results for RQ2. Table 8 confirms the conflicting nature between several of the quality attributes and most of the security metrics by analyzing the impact of the refactoring solutions generated by our tool on the 30 open source projects. Four of the quality attributes were negatively correlated with the security metrics except Flexibility and Effectiveness. Reusability and Extendibility are negatively correlated with most of the security metrics which confirms the results of RQ1. In fact, these quality attributes can be improved using the extract super/sub class and pull-up method/field refactoring types that were already negatively correlated with security metrics.

Figure 7 presents more details related to the distribution of the refactoring solutions on the 30 open source projects based on each pair of quality and security metrics (all the metrics are to minimize based on our formulation). The distribution of the solutions is consistent with the correlation results reported in Table 8. For instance, the refactoring solutions with good reusability (low values) have the worst security impacts (high values) on the open source projects.

Since it is not enough to check the ability of our refactoring solutions to improve the quality and security objectives, we asked the 15 selected participants to evaluate the generated refactorings for 5 of the 30 open source projects using our tool (+Security) and an existing refactoring tool (-Security) [9]. The average manual correctness on the five systems is 86% for our approach compared to 73% for [9] (without the consideration of security objective) as described in Figure 8. Thus, it is clear that refactoring solutions addressing both quality and security issues were preferred compared to only improving the quality metrics. We presented the refactorings in a random way (not on the same code locations) to the participants and they were not aware of which tool is used to generate them. The refactorings recommended for the Gantt project were all considered relevant by the participants. The obtained results confirm that the combination of both quality and security objectives reasonably match the preferences of the participants.

Finding 2: Understandability, Reusability, Functionality and Extendibility are all negatively correlated with the avg security metric. Flexibility and Effectiveness are positively correlated with the avg security metric. Reusability and Extendibility are negatively correlated with all of the eight security metrics.

Results for RQ3. Figure 9 summarizes the comparison of our tool with the work of Alizadeh et al. [9], not considering the security objective. The goal is to understand the sacrifice in quality when improving the security objective using the generated refactoring solutions. While Alizedeh et al.'s tool [9] improved the quality attributes more than our tool, the improvements are very similar to our security-aware approach for almost all the quality metrics. The major difference is for the extendibility measure, which is

TABLE 8: Correlation results between the average of security metrics and quality attributes on the 30 projects. The results are statistically significant using the two-sample t-test at a 95% confidence level (α = 5%)

QMOOD / Security Metrics	Understandability	Reusability	Functionality	Flexibility	Extendibility	Effectiveness
CIDA	- (-0.237)	- - (-0.617)	- (-0.184)	- (-0.318)	- (-0.391)	+ (0.116)
CCDA	- (-0.224)	- (-0.382)	- (-0.137)	+ (0.281)	- (-0.232)	+ + (0.589)
COA	+ (0.192)	- (-0.373)	- (-0.183)	+ (0.219)	- - (-0.619)	+ (0.314)
CMAI	- (-0.217)	- (-0.387)	- (-0.120)	+ (0.113)	- (-0.382)	+ (0.221)
CAAI	+ (0.114)	- (-0.234)	+ (0.131)	++ (0.612)	- (-0.224)	- (-0.122)
CAIW	- (-0.213)	- (-0.346)	- (-0.114)	+ (0.116)	- - (0.563)	+ (0.138)
CMW	+ (0.194)	- (-0.213)	- (-0.233)	+ (0.221)	- (0.241)	+ (0.187)
VA	- (-0.226)	- (-0.362)	- (-0.341)	+ (0.412)	- (-0.268)	+ (0.224)
AvgSecurity	- (-0.382)	- - (-0.731)	- (- 0.114)	+ (0.183)	- - (-0.618)	+ (0.213)

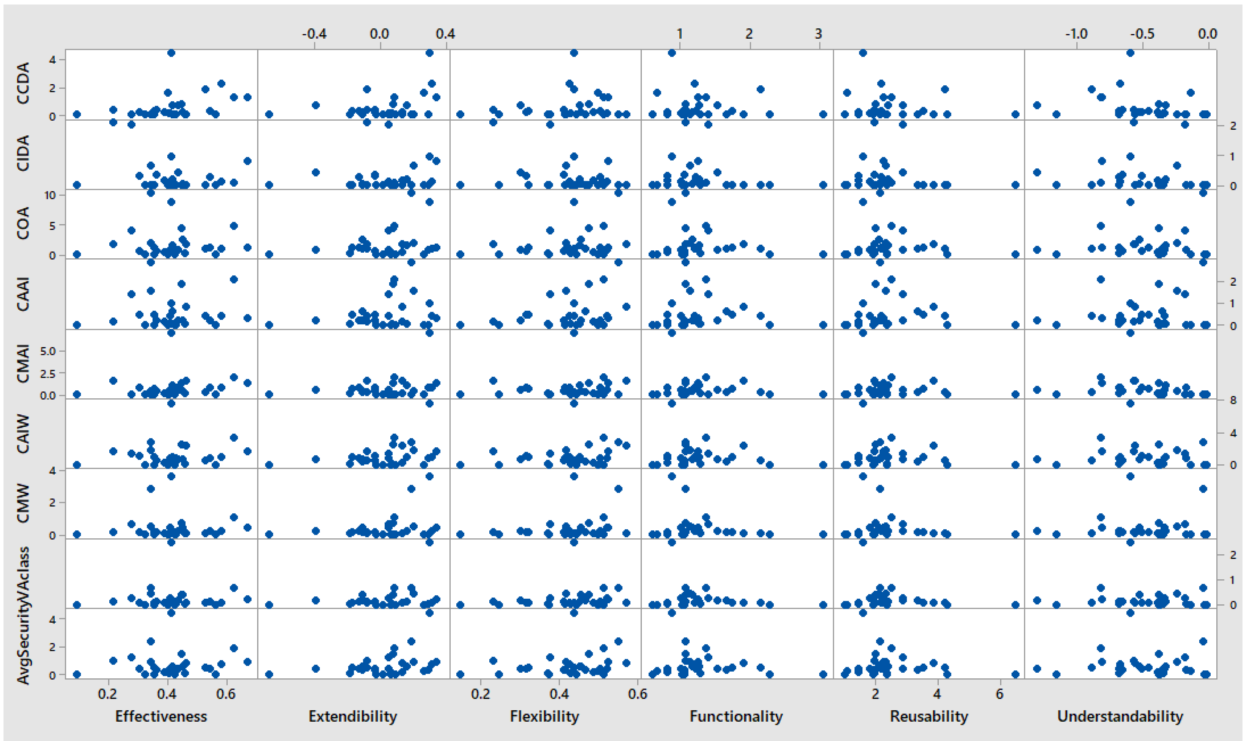


Fig. 7: Distribution of refactoring solutions based on each pair of quality and security metrics for the 30 projects.

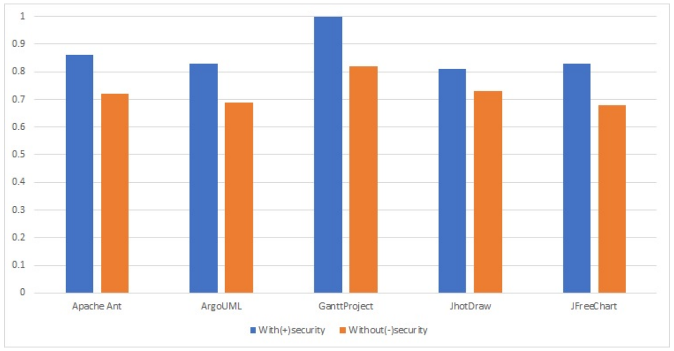


Fig. 8: Average manually determined correctness of the refactorings on different open source projects generated by our tool (+Security) and an existing refactoring tool (-Security) [9]..

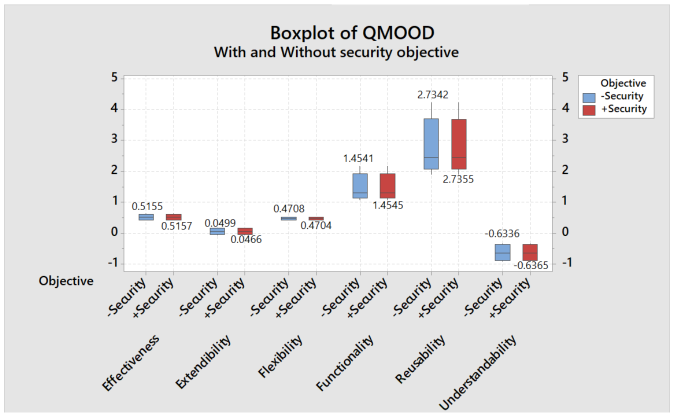


Fig. 9: Box plots of the impact of refactoring solutions on the quality attributes based on 4 open source projects using our tool (+Security) and an existing refactoring tool (-Security) [9]. The results are statistically significant using the two-sample t-test at a 95% confidence level (α = 5%)

understandable based on the results of RQ1 and RQ2, and the difference is rather small.

Figure 10 shows that the multi-objective security-aware

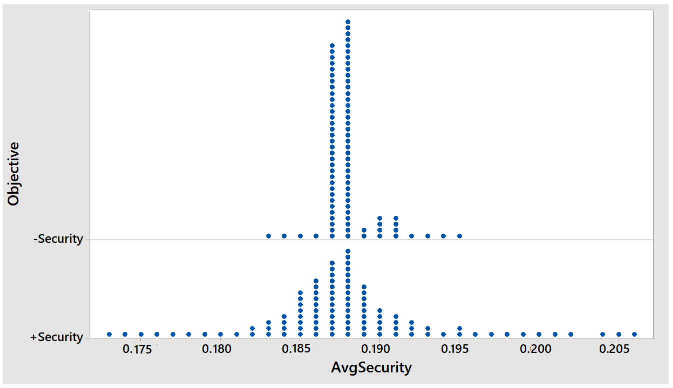


Fig. 10: Distribution of the refactoring solutions using the security objective based on 4 open source projects comparing our tool (+Security) and an existing refactoring tool (-Security) [9].

approach was able to generate a diverse set of refactoring solutions in terms of security improvements. The tool of Alizadeh et al. [9] was not able to generate any refactoring solution that can have a security objective value lower than 0.183 while our approach was able to improve better the security metric to reach lower than 0.175. While the deviation in terms of value may look small, the formulation of the security objective actually requires significant code changes to slightly improve security values.

> *Finding 3:* The sacrifice, by our approach, in terms of quality improvements is very limited when enhancing code security comparing to an existing work only based on quality [9].

Results for RQ4. We asked participants to rate their agreement on a Likert scale from 1 (complete disagreement) to 5 (complete agreement) with the following questions:

- The security-aware refactoring recommendations are a desirable feature in integrated development environments to improve code security while enhancing quality.
- The security-aware refactoring web app is easy to use compared to fully-automated or manual refactoring tools that you used in the past.

The post-study questionnaire results show the average agreement of the participants was 3.96 and 4.12 based on a Likert scale for the first and second statements, respectively. This confirms both the relevance and usability of our security-aware tool to find a trade-off between code security and quality metrics. More details can be found in our appendix [21] showing the simple steps developers can follow to evaluate and fix both the quality and security issues of their projects.

We also asked the participants about the most important reasons to refactor their code. Figure 11 shows, surprisingly, that most of the participants considered security as *the most critical reason* for refactoring, even compared to improving quality metrics which is the second most important motivation for refactoring. Bug likelihood and code smells were also considered important by some participants. The outcomes of this question on why to refactor the code are aligned with the motivations of this paper advocating for considering both security and quality metrics when recommending refactorings.

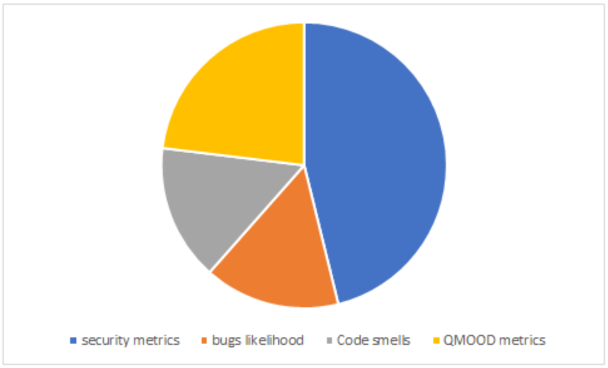


Fig. 11: The important motivations for code refactoring by the participants.

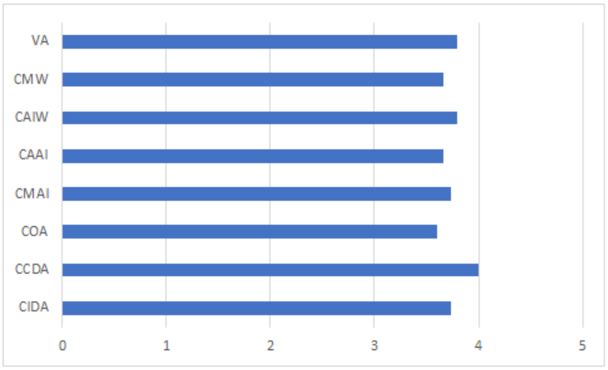


Fig. 12: The potential impacts of refactoring on security metrics based on the survey.

The next questions asked the respondents about the impact of refactoring on the code security metrics. Figure 12 shows that the developers think that refactoring can improve and positively impact most of the security metrics considered in our experiments. This confirms our selection of the security metrics and the outcomes of RQ1 obtained by analyzing the code. The developers think that the CCDA metric is the one that can be most improved by refactoring. The CCDA

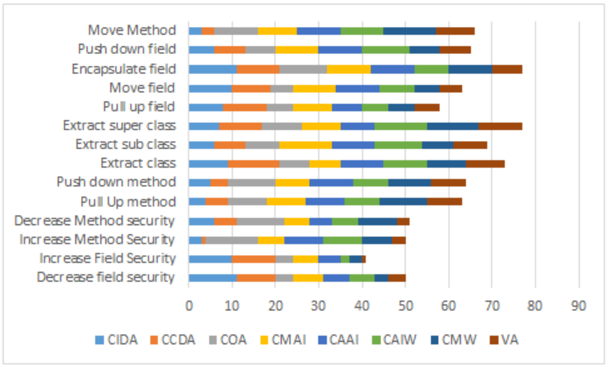


Fig. 13: The potential impact of different refactoring types on security metrics based on the survey.

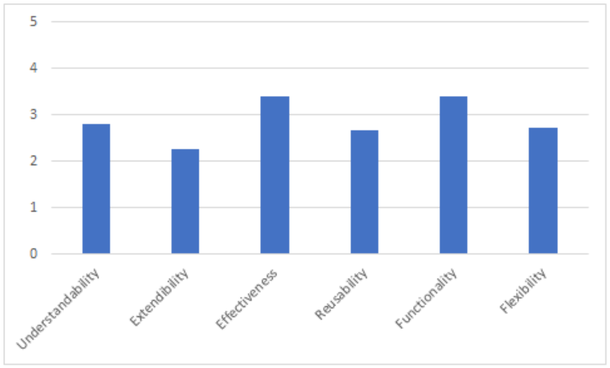


Fig. 14: The possible positive impact of improving the security metrics on quality attributes based on the survey.

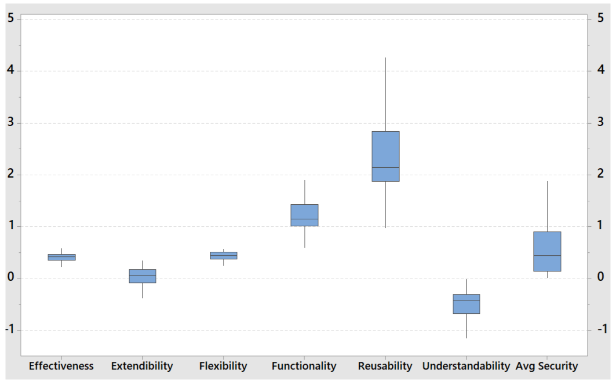


Fig. 15: Box plots of the impacts of refactoring solutions on both quality and security objectives based on the 30 projects.

metric measures the direct access of classified class attributes of a class. It aims to protect the internal representations of a class, i.e. class attributes, from direct access. In fact, the accessibility of class attributes is one of the most critical entry points for security attacks to the architecture, and so the use of refactorings such as Increase Field Security can improve this metric. Figure 13 describes more detailed results on the possible impact of each refactoring type on the various static code security metrics. It is clear that Encapsulate field can have the most positive impact on several security metrics based on the developers' feedback. They also suggested that Extract Superclass will impact the security metrics, but in a negative way. The participants found as well that Push down method refactoring can improve several security metrics since it will reduce accessibility to the methods after refactoring. The results of these questions also confirm the results obtained in RQ1 about the impact of different refactoring types on security when we analyzed the code before and after refactoring.

Figure 14 shows the opinion of developers on whether improving the security metrics will positively impact some quality attributes. The results show that effectiveness and functionally quality attributes can be improved if the refactorings improved security. The developers also suggested that improving security will have a negative impact on both understandability and extendibility since they have the least support from developers (around 2.7 out of 5). These outcomes are also partially consistent with the results found in RQ2 when we analyzed the correlation between the security metrics and quality attributes based on the code level information before and after refactoring.

Figure 15 shows that our approach based on multi-objective search can find good trade-offs between the various quality and security objectives. The box plots describe the diversity of the refactoring solutions generated by our multi-objective approach where the developer can find solutions that impact both quality and security at different levels. This aspect is important since a developer can select solutions that impact their specific quality or security objectives based on their preferences.

The impact of the refactorings on the different quality and security metrics is calculated based on the differences between their values before and applying the refactorings. The formulas of the different metrics are described in Tables 1 and 2. Thus, we just measured the difference of these metric values before and after applying the refactorings to estimate the improvements. The box plots show that the generated refactorings can improve the majority of the quality and security objectives with varying levels of improvement, but sometimes it is possible to deteriorate (or sacrifice) some of the metric values/improvements due to their conflicting nature. However, Figure 15 shows that the multi-objective algorithm was able to generate solutions improving the objectives at different levels with little deterioration. Thus, we conclude that the tool was successful in finding trade-offs rather than merely improving one or two specific objectives.

To summarize, the participants found the tool unique in terms of enabling them to understand the impact of refactoring on both security and quality. They highlighted that it is one of the first tools in their opinion that enables the identification of refactoring solutions to offer trade-offs between quality and security. The developers found the tool flexible as it provides multiple options to select a solution based on their preferences. A suggested improvement is to use visualization techniques to evaluate the impact of applying a refactoring sequence on the different security and quality metrics.

Finding 4: The evaluation of our tool by 15 developers confirmed its efficiency in helping to understand the impact of refactoring on both security and quality and generating refactoring solutions that find a trade-offs between quality and security.

5 Threats To Validity

The parameter tuning of the NSGA-II optimization algorithm used in our experiments is the first internal threat since these values were found by trial-and-error [41]. Since we used a limited number of evaluated systems and participants, the generalizability of our results is threatened. Besides, we only considered 14 refactoring types in our study. Furthermore, for the manual validation and comparison with an existing refactoring study, we used a selected subset of projects rather than the full 30 systems. Therefore, we estimate that a potential replication of our work is necessary to validate our results completely. We are also planning to consider more security and quality metrics to extend our empirical validation. The opinions of the practitioners involved in our study may be divergent when it comes to the recommended refactorings, and they might have different priorities for the security of the system which could have an impact on our results. Furthermore, our security metrics are limited to 8 measures thus we may need to include further metrics in our future studies and not only the easiest ones to implement.

Another potential threat is related to the identification of security sensitive attributes which can impact the calculation of the security metrics. To mitigate this threat, we manually validated the top 10 critical files and use their critical attributes (fields that have names that match one of the keywords from the list we gathered at the beginning) to identify the critical attributes in all the other files that will be used to compute the security metrics.

Finally, there is a possible threat due to experimenter bias in the surveys as the subjects had some prior contact with the researchers.

6 Related Work

We first review studies dealing with refactoring as a search problem [42], [43], [44], [45], [46], [47], [48], [49], [50], [51], [52], [53], [54]. Then, we focus on studies investigating refactoring for security purposes.

6.1 Search-Based Refactoring

O'Keeffe and Cinnéide [55] presented the idea of formulating the refactoring task as a search problem in the space of alternative designs, generated by applying a set of refactoring operations. The search is guided by a quality evaluation function based on eleven object-oriented design metrics that reflect refactoring goals. Ouni et al. [12] tried to find recommendations that tend to maximize the use of refactoring rules applied in the past to similar contexts from one side, and to minimize semantic errors and the number of defects from another. In another work [10], they focused on refactoring solutions minimizing the number of bad-smells while maximizing the use of development history and semantic coherence.

Alizadeh et al. [9] generated refactoring solutions that optimize the QMOOD metrics while minimizing the deviation from the initial design. In another work [11], they considered the QMOOD metrics as objectives for their optimization problem. Then, they used an unsupervised learning algorithm to cluster the different trade-off solutions in order to reduce the developers' interaction effort when refactoring systems. Harman and Tratt [26] were the first to introduce the concept of Pareto optimality to search-based refactoring. They used it to combine two metrics, namely CBO (Coupling Between Objects) and SDMPC (Standard Deviation of Methods Per Class), into a fitness function and showed its superior performance as compared to a mono-objective technique [26].

None of the work mentioned has directly addressed the research questions we investigated here: trying to find refactoring solutions while considering the code quality attributes and security metrics as conflicting objectives.

6.2 Security-Aware Refactoring

Alshayeb et al. [56] proposed an empirical study to check the relationships between some types of code smell, such as feature envy, and security metrics. The results show that these code smell did not negatively impact the security metrics. In this paper, we did not focus on code smells but more on the impact of refactorings and improving quality attributes on security.

Maruyama et al. [13], [14] implemented a prototype of an automated refactoring tool detecting possible code vulnerabilities. The tool presents the programmers with information on the security level of the modified code. However, their tool, Jsart, supports only two refactorings (Push up method and Push down method) and assesses the decrease of the security level by the decrease of the access levels after the application of the above refactorings. Thus, they do not use metrics as an assessment tool.

Alshammari et al. [15] studied the impact of refactoring rules on the security of an object-oriented design using the security design metrics [18], [19], [57], [58]. They also introduced new security refactoring rules per analogy to existing ones and distinguished their effects on classified and non-classified features. They proposed one case study to illustrate how applying the refactoring rules improves the security of the design. Therefore, their findings are not general.

Ghaith and Cinnéide [16] presented an approach to automated improvement of software security based on search-based refactoring using the Code-Imp platform. When this platform is used to improve software design, the fitness function is a combination of quality metrics. In their work, they redefined this fitness function based uniquely on security metrics. Therefore, they neither studied the relationship between security and quality, nor the impact of the security-aware refactorings on the quality of the system. They also looked at the impact of certain refactorings on the security metrics, but since they considered just one study case, their results cannot be generalized.

7 Conclusion

We have presented an empirical study to validate the correlations between the QMOOD quality attributes [17] and a set of security metrics [18], [19] and to understand the correlations between refactoring types and security metrics. Based on the outcomes of these studies, we proposed a security-aware multi-objective refactoring approach to find a balance between quality and security goals. We evaluated our tool on the same projects used for the empirical validations. Furthermore, we compared our results to an existing refactoring work not considering security to understand the sacrifice in security measures when improving the quality. The comparison shows that our security-aware approach performed significantly better than the existing approach when it comes to preserving and improving the security of the system but with low cost in terms of sacrificing quality. The survey with the 15 practitioners confirmed the efficiency of our tool and the importance of considering security while improving several quality attributes.

We are planning as part of our future work to expand the set of supported security metrics to include design-level metrics [57], [58], [59] as well, in a similar study. We are also planning to study the correlation between security metrics and the impact of improving one on the other. We are planning to expand our set of refactorings by those that can change the relationship between classes, such as Replace Inheritance with Delegation. It is an accepted principle in industry that a delegation relationship should be preferred to inheritance, particularly in the context of inversion of control containers such as Spring. Thus, we are planning to study the impact of these new types of refactoring on security and quality then check their acceptability by developers. Another research direction would be to generate refactoring recommendations that include third-party libraries [60], [61] in order to understand their impact on the security of JAVA apps. Finally, we are planning to perform a survey with developers to investigate the importance of considering security as a goal/motivation for refactoring.

REFERENCES

[1] Nist and E. Aroms, *NIST Special Publication 800-53 Revision 3 Recommended Security Controls for Federal Information Systems and Organizations*. Paramount, CA: CreateSpace, 2012.

[2] S. Planning, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology*, 2002.

[3] W. Suryn, A. Abran, and A. April, "Iso/iec square. the second generation of standards for software product quality," 2003.

[4] T. A. Linden, "Operating system structures to support security and reliable software," *ACM Computing Surveys (CSUR)*, vol. 8, no. 4, pp. 409–445, 1976.

[5] A. Adewumi, S. Misra, and N. Omoregbe, "Evaluating open source software quality models against iso 25010," in *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*. IEEE, 2015, pp. 872–877.

[6] A. Przybylek, "Impact of aspect-oriented programming on software modularity," in *2011 15th European Conference on Software Maintenance and Reengineering*. IEEE, 2011, pp. 369–372.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, MA, USA, 1999.

[8] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, "Search-based refactoring: Towards semantics preservation," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 347–356.

[9] V. Alizadeh, M. Kessentini, W. Mkaouer, M. Ocinneide, A. Ouni, and Y. Cai, "An interactive and dynamic search-based approach to software refactoring recommendations," *IEEE Transactions on Software Engineering*, 2018.

[10] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, "The use of development history in software refactoring using a multi-objective evolutionary algorithm," in *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. ACM, 2013, pp. 1461–1468.

[11] V. Alizadeh and M. Kessentini, "Reducing interactive refactoring effort via clustering-based multi-objective search," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 464–474.

[12] A. Ouni, M. Kessentini, and H. Sahraoui, "Search-based refactoring using recorded code changes," in *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 2013, pp. 221–230.

[13] K. Maruyama and K. Tokoda, "Security-aware refactoring alerting its impact on code vulnerabilities," in *2008 15th Asia-Pacific Software Engineering Conference*, Dec 2008, pp. 445–452.

[14] K. Maruyama and T. Omori, "A security-aware refactoring tool for java programs," 01 2011.

[15] B. Alshammari, C. Fidge, and D. Corney, "Assessing the impact of refactoring on security-critical object-oriented designs," in *2010 Asia Pacific Software Engineering Conference*, Nov 2010, pp. 186–195.

[16] S. Ghaith and M. Ó Cinnéide, "Improving software security using search-based refactoring," in *Search Based Software Engineering*, G. Fraser and J. Teixeira de Souza, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 121–135.

[17] P. K. Goyal and G. Joshi, "Qmood metric sets to assess quality of java program," in *2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*. IEEE, 2014, pp. 520–533.

[18] A. Agrawal and R. Khan, "Assessing impact of cohesion on security-an object oriented design perspective," *Pensee*, vol. 76, no. 2, 2014.

[19] B. Alshammari, C. Fidge, and D. Corney, "Security metrics for object-oriented class designs," in *2009 Ninth International Conference on Quality Software*. IEEE, 2009, pp. 11–20.

[20] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[21] TSE. (2020) Online appendix for this publication. *https://doi.org/10.7302/0bgn-vt27* .

[22] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4–17, 2002.

[23] 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.

[24] A. C. Jensen and B. H. Cheng, "On the use of genetic programming for automated refactoring and the introduction of design patterns," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*. ACM, 2010, pp. 1341–1348.

[25] S. Lee, G. Bae, H. S. Chae, D.-H. Bae, and Y. R. Kwon, "Automated scheduling for clone-based refactoring using a competent ga," *Software: Practice and Experience*, vol. 41, no. 5, pp. 521–550, 2011.

[26] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. ACM, 2007, pp. 1106–1113.

[27] R. Shatnawi and W. Li, "An empirical assessment of refactoring impact on software quality using a hierarchical quality model," *International Journal of Software Engineering and Its Applications*, vol. 5, no. 4, pp. 127–149, 2011.

[28] M. O'Keeffe and M. O. Cinnéide, "Search-based refactoring: An empirical study," *Journal of Software Maintenance and Evolution*, vol. 20, no. 5, pp. 345–364, 2008.

[29] S. A. Ansar, R. A. Khan *et al.*, "A phase-wise review of software security metrics," in *Networking Communication and Data Knowledge Engineering*. Springer, 2018, pp. 15–25.

[30] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for java program analysis: a retrospective," in *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, vol. 15, 2011, p. 35.

[31] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics vs text mining," in *2014 IEEE 25th international symposium on software reliability engineering*. IEEE, 2014, pp. 23–33.

[32] B. Alshammari, C. Fidge, and D. Corney, "Security metrics for object-oriented class designs," in *2009 Ninth International Conference on Quality Software*. IEEE, 2009, pp. 11–20.

[33] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, 2014.

[34] Y. Tang, F. Zhao, Y. Yang, H. Lu, Y. Zhou, and B. Xu, "Predicting vulnerable components via text mining or software metrics? an effort-aware perspective," in *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 2015, pp. 27–36.

[35] J. Jürjens, *Secure systems development with UML*. Springer Science & Business Media, 2005.

[36] N. Rachatasumrit and M. Kim, "An empirical investigation into the impact of refactoring on regression testing," in *2012 28th Ieee International Conference on Software Maintenance (Icsm)*. IEEE, 2012, pp. 357–366.

[37] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on software engineering*, vol. 28, no. 1, pp. 4–17, 2002.

[38] W. F. Opdyke, "Refactoring object-oriented frameworks," 1992.

[39] J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," in *Noise reduction in speech processing*. Springer, 2009, pp. 1–4.

[40] S. P. Coy, B. L. Golden, G. C. Runger, and E. A. Wasil, "Using experimental design to find effective parameter settings for heuristics," *Journal of Heuristics*, vol. 7, no. 1, pp. 77–97, 2001.

[41] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 11:1–11:61, Dec. 2012. [Online]. Available: http://doi.acm.org/10.1145/2379776.2379787

[42] M. Boussaa, W. Kessentini, M. Kessentini, S. Bechikh, and S. B. Chikha, "Competitive coevolutionary code-smells detection," in *International Symposium on Search Based Software Engineering*. Springer, Berlin, Heidelberg, 2013, pp. 50–65.

[43] A. Ghannem, M. Kessentini, and G. El Boussaidi, "Detecting model refactoring opportunities using heuristic search," in *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, 2011, pp. 175–187.

[44] S. Kalboussi, S. Bechikh, M. Kessentini, and L. B. Said, "Preference-based many-objective evolutionary testing generates harder test cases for autonomous agents," in *International Symposium on Search Based Software Engineering*. Springer, Berlin, Heidelberg, 2013, pp. 245–250.

[45] J. Shelburg, M. Kessentini, and D. R. Tauritz, "Regression testing for model transformations: A multi-objective approach," in *International Symposium on Search Based Software Engineering*. Springer, Berlin, Heidelberg, 2013, pp. 209–223.

[46] M. Kessentini, R. Mahaouachi, and K. Ghedira, "What you like in design use to correct bad-smells," *Software Quality Journal*, vol. 21, no. 4, pp. 551–571, 2013.

[47] A. Ghannem, G. El Boussaidi, and M. Kessentini, "Model refactoring using examples: a search-based approach," *Journal of Software: Evolution and Process*, vol. 26, no. 7, pp. 692–713, 2014.

[48] M. Kessentini, A. Ouni, P. Langer, M. Wimmer, and S. Bechikh, "Search-based metamodel matching with structural and syntactic measures," *Journal of Systems and Software*, vol. 97, pp. 1–14, 2014.

[49] B. Amal, M. Kessentini, S. Bechikh, J. Dea, and L. B. Said, "On the use of machine learning and search-based software engineering for ill-defined fitness function: a case study on software refactoring," in *International Symposium on Search Based Software Engineering*. Springer, Cham, 2014, pp. 31–45.

[50] M. W. Mkaouer, M. Kessentini, S. Bechikh, and M. Ó. Cinnéide, "A robust multi-objective approach for software refactoring under uncertainty," in *International Symposium on Search Based Software Engineering*. Springer, Cham, 2014, pp. 168–183.

[51] U. Mansoor, M. Kessentini, P. Langer, M. Wimmer, S. Bechikh, and K. Deb, "Momm: Multi-objective model merging," *Journal of Systems and Software*, vol. 103, pp. 423–439, 2015.

[52] A. Ghannem, G. El Boussaidi, and M. Kessentini, "On the use of design defect examples to detect model refactoring opportunities," *Software Quality Journal*, vol. 24, no. 4, pp. 947–965, 2016.

[53] H. Wang, M. Kessentini, and A. Ouni, "Bi-level identification of web service defects," in *International Conference on Service-Oriented Computing*. Springer, Cham, 2016, pp. 352–368.

[54] A. Ouni, M. Kessentini, M. Ó Cinnéide, H. Sahraoui, K. Deb, and K. Inoue, "More: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells," *Journal of Software: Evolution and Process*, vol. 29, no. 5, p. e1843, 2017.

[55] M. O'Keeffe and M. Ó. Cinnéide, "A stochastic approach to automated design improvement," in *Proceedings of the 2nd international conference on Principles and practice of programming in Java*. Computer Science Press, Inc., 2003, pp. 59–62.

[56] 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.

[57] B. Alshammari, C. Fidge, and D. Corney, "Security metrics for object-oriented class designs," in *2009 Ninth International Conference on Quality Software*, Aug 2009, pp. 11–20.

[58] G. McGraw and E. W. Felten, *Securing Java: Getting Down to Business with Mobile Code*. New York, NY, USA: John Wiley & Sons, Inc., 1999.

[59] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Decoupling level: a new metric for architectural maintenance complexity," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 499–510.

[60] M. Reif, M. Eichberg, B. Hermann, and M. Mezini, "Hermes: assessment and creation of effective test corpora," in *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, 2017, pp. 43–48.

[61] M. Reif, M. Eichberg, B. Hermann, J. Lerch, and M. Mezini, "Call graph construction for java libraries," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 474–486.

Chaima Abid is currently a PhD student in the intelligent Software Engineering group at the University of Michigan. Her PhD project is concerned with the application of intelligent search and machine learning in different areas such as web services, refactoring and security. Her current research interests are Search-Based Software Engineering, web services, refactoring, security, data analytics and software quality.

Marouane Kessentini is a recipient of the prestigious 2018 President of Tunisia distinguished research award, the University distinguished teaching award, the University distinguished digital education award, the College of Engineering and Computer Science distinguished research award, 4 best paper awards, and his AI-based software refactoring invention, licensed and deployed by industrial partners, is selected as one of the Top 8 inventions at the University of Michigan for 2018 (including the three campuses), among over 500 inventions, by the UM Technology Transfer Office. He is currently a tenured associate professor and leading a research group on Software Engineering Intelligence. Prior to joining UM in 2013, He received his Ph.D. from the University of Montreal in Canada in 2012. He received several grants from both industry and federal agencies and published over 110 papers in top journals and conferences. He has several collaborations with industry on the use of computational search, machine learning and evolutionary algorithms to address software engineering and services computing problems.

Vahid Alizadeh is currently a Ph.D. student in the intelligent Software Engineering group at the University of Michigan. His Ph.D. project is concerned with the application of intelligent search and machine learning in different software engineering areas such as refactoring, testing, and documentation. His current research interests are Search-Based Software Engineering, Refactoring, Artificial Intelligence, data analytics and software quality.

Mouna Dhaouadi is a master student in the intelligent Software Engineering group at the University of Michigan. Her primary research interests are Search-Based Software Engineering, security, and refactoring.

Rick Kazman is a Professor at the University of Hawaii and a Principal Researcher at the Software Engineering Institute of Carnegie Mellon University. His primary research interests are software architecture, design and analysis tools, software visualization, and software engineering economics. He also has interests in human-computer interaction and information retrieval. Kazman has created several highly influential methods and tools for architecture analysis, including the SAAM (Software Architecture Analysis Method), the ATAM (Architecture Tradeoff Analysis Method), the CBAM (Cost-Benefit Analysis Method) and the Dali architecture reverse engineering tool.