Automated Software Engineering (2021) 28:4
https://doi.org/10.1007/s10515-021-00281-2

Prioritizing refactorings for security-critical code

Chaima Abid[1] · Vahid Alizadeh[1] · Marouane Kessentini[1] · Mouna Dhaouadi[1] · Rick Kazman[2]

Received: 5 February 2020 / Accepted: 5 March 2021 / Published online: 18 May 2021
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2021

Abstract

It is vitally important to fix quality issues in security-critical code as they may be sources of vulnerabilities in the future. These quality issues may increase the attack surface if they are not quickly refactored. In this paper, we use the history of vulnerabilities and security bug reports along with a set of keywords to automatically identify a project's security-critical files based on its source code, bug reports, pull-request descriptions and commit messages. After identifying these security-related files, we estimate their risks using static analysis to check their coupling with other project components. Then, our approach recommends refactorings to prioritize fixing quality issues in these security-critical files to improve quality attributes and remove identified code smells. To find a trade-off between the quality issues and security-critical files, we adopted a multi-objective search strategy. We evaluated our approach on six open source projects and one industrial system to check the correctness and relevance of the refactorings targeting security critical code. The results of our survey with practitioners supports our hypothesis that quality and security need to be considered together to provide relevant refactoring recommendations.

Keywords Refactoring · Security · Recommendations

✉ Marouane Kessentini
marouane@umich.edu

Chaima Abid
cabid@umich.edu

Vahid Alizadeh
alizadeh@umich.edu

Mouna Dhaouadi
mounad@umich.edu

Rick Kazman
kazman@hawaii.edu

[1] University of Michigan-Dearborn, Dearborn, USA

[2] University of Hawaii, Honolulu, USA

1 Introduction

The National Institute of Standards and Technology (NIST) estimates that the US economy loses an average of $60 billion per year as a cost of either implementing patches to fix security bugs and vulnerabilities or the actual impact of these security issues (Cusumano 2004). Vulnerability is defined as a property of system security requirements, design, implementation, or operation that could be accidentally or intentionally exploited to create a security failure (Krsul 1998). These vulnerabilities heavily depend on the way a system is designed and implemented. For instance, many software companies use third-party code and libraries (Nikiforakis et al. 2012) and many vulnerabilities are introduced through these external components (Han and Zheng 1998). Thus, it is critical to identify the security-critical code fragments when integrating new modules or to locate them in internally developed code to protect the system against possible attacks. Security-critical code refers to code fragments that contain data (e.g., attributes) and logic (e.g., methods) that can potentially be misused to violate security properties such as confidentiality, integrity, or availability of a system in production.

Several studies on the detection and fixing of vulnerabilities and security bugs (Livshits and Lam 2005; Haldar et al. 2005) show that poor quality indicators are one of the main sources of vulnerabilities, as also emphasized by CWE (CWE-398) (Huang et al. 2016). The Common Weakness Enumeration (CWE)[1] is a category system for software weaknesses and vulnerabilities. It is sustained by a community project with the goals of understanding flaws in software and creating automated tools that can be used to identify, fix, and prevent those flaws. CWE-398 refers to the types of software weaknesses that are "Indicators of Poor Code Quality". This provides additional evidence that code quality issues are frequently responsible for security issues. The description of this category highlights that when the code is complex and not well-maintained it is more likely to cause security problems and weaknesses. However, existing refactoring research is mainly focused on improving quality attributes and fixing code smells (Ouni et al. 2013a, 2015, 2017; Mkaouer et al. 2014a, 2017). For instance, a developer may create a hierarchy in a set of classes to improve the reusability quality attribute. However, these actions may expand the attack surface if the super class contains security-critical attributes and methods. Furthermore, the few existing studies on the prioritization of refactorings mainly focus on the identified quality issues but without considering security as one of the criteria, despite its importance and relevance in practice (Tsantalis and Chatzigeorgiou 2011; Zazworka et al. 2011; Vidal et al. 2016a). Without a direct focus on security related quality issues, the ranking schemes of related work can potentially ignore critical security issues because they may be given lower ranks.

In this paper, we used the history of vulnerabilities and security bug reports along with a set of keywords [defined in the literature (Alshammari et al. 2009, 2010a)] to automatically identify security-critical files in a project based on source code,

[1] http://cwe.mitre.org/about/index.html.

bug reports, pull-request descriptions and commit messages. After identifying these security-related files, we estimated their risk based on static analysis to check their coupling with other components of the project. For instance, a highly coupled class which contains security-critical code fragments may contribute to compromising the whole system if an attacker takes advantage of the code to inject malicious payloads. Then, our approach recommends refactorings to prioritize fixing quality issues in these security-critical files to improve quality attributes and remove identified code smells. To find a trade-off between the quality issues and security-critical files, we adopted a multi-objective search (Deb et al. 2002) approach.

We evaluated our approach on six open source projects and one industrial system to check the relevance of our refactoring recommendations. The results confirm the effectiveness of our approach comparing to existing refactoring studies based on quality attributes and ranking the recommendations only based on their code smells and quality severity (Ouni et al. 2016; Fokaefs et al. 2011). Our survey with practitioners who used our tool supports our hypothesis that quality and security need to be considered together to provide relevant refactoring recommendations and to rank them.

The remainder of this paper is organized as follows: Sect. 2 discusses motivations that inspired this research. Section 3 presents the description of our approach while Sect. 4 contains the experiments on 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 Sect. 7.

2 Motivations and challenges

Security-critical code fragments in a software project can represent code elements (e.g. classes, methods, files) containing confidential or sensitive information such as IDs, transactions, credit card data, authentication information, and security constraints. If these code fragments are over-exposed then they may result in vulnerabilities that may be exploited in violating security properties (Abid et al. 2020). Code fragments are frequently cited in security bug reports, vulnerability reports, or Stack Overflow posts which suggests that they are at the heart of many security problems. And often vulnerable code fragments, identified during code reviews, are analyzed to ensure that they are carefully designed so as to reduce the attack surface in case of potential attacks in the future (Abid et al. 2020).

The identification of security-relevant code in a software project is critical (1) for designers to be careful when they are designing or maintaining a system. For instance, they have to make sure that coupling is low in these security-related fragments to reduce the attack surface; (2) for developers to ensure that these code fragments are not over-exposed; (3) for reviewers to pay a lot of attention when reviewing these files; and (4) for the organization to evaluate the use of third-party code from a security perspective before any adoption or integration work. However, most existing research tools for refactoring recommendation and prioritization (Ouni et al. 2013a, 2015, 2017; Mkaouer et al. 2014a, 2017) do not consider the security aspect

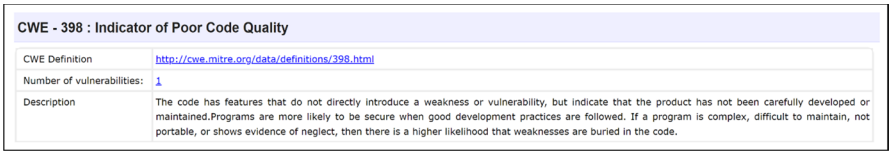

Fig. 1 A type of software weakness from the CWE list (CWE 2009) that includes security vulnerabilities related to poor code quality

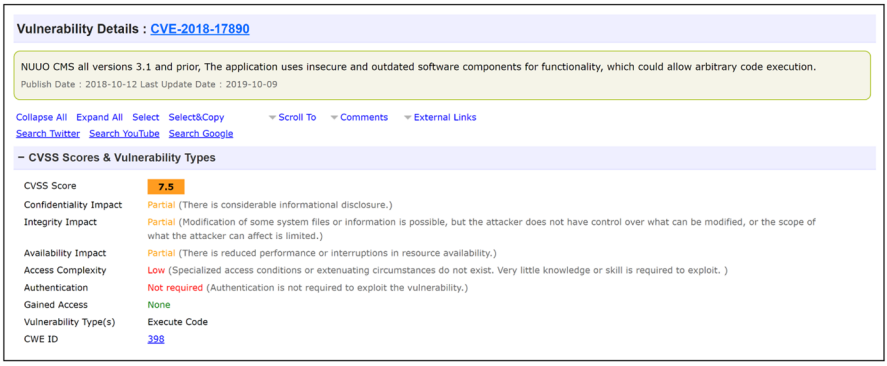

Fig. 2 An example of a security vulnerability from NUUO CMS system due to code quality issues (Nuuo 2018)

but focus more on general quality improvements and removal of code smells when ranking and recommending refactorings.

Nowadays, maintaining both quality and security of software systems is not optional. Many contemporary applications are cloud-based and therefore potentially exposed to malicious attacks. Developers are under increasing pressure to deliver clean and reliable software systems that generate the intended outputs while making sure that sensitive customers data is secure.

One of the main challenges when integrating both code quality and security concerns into a single refactoring tool is that they may be conflicting. For example, improving the reusability of the code may increase the attack surface due to newly created abstractions. Also, increasing the spread of classes that contain sensitive information in the design to improve modularity may reduce the resilience of the system to attacks.

The Common Vulnerabilities and Exposures database (CVE) is a large, publicly available source of vulnerability reports (Cve 2021). It aims to provide common names for publicly known problems. As described in Fig. 1, one of the main CVE categories is "Indicator of Poor Code Quality" (CWE-398) providing additional evidence that code quality issues are frequently responsible for security issues. The description of this category highlights that when the code is complex and not well-maintained it is more likely to cause security problems and weaknesses.

Figure 2 shows an example of one detected vulnerability in the CWE-398 category on the NUUO Intelligent Surveillance software system (Nuuo 2018) due to the

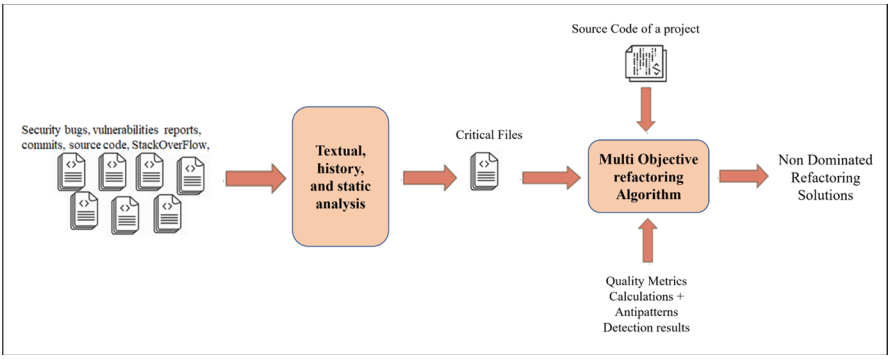

Fig. 3 Security-critical code identification: approach overview

use of multiple outdated software components that needed to be refactored. Whenever developers introduced changes to the system, they faced challenges to make those changes consistently across classes; thus introducing refactoring to fix this quality issue was critical. This vulnerability had a score of 7.5 which is considered to be high and urgent to fix.

To overcome such challenges, in the next section we propose an approach to prioritize and recommend refactorings to target the classes that have both quality and security issues. The goal is to enable developers to spend less effort on refactoring non-critical issues and make systems more secure while maintaining high code quality.

3 Approach

The structure of our approach is sketched in Fig. 3. The first component consists of identifying a project's security-critical files, to evaluate and refactor, based on a list of keywords, along with the history of security bugs and vulnerabilities detected in the analyzed system. The list of keywords are the most common security-related words that developers may use in naming code elements, writing comments, security bugs and vulnerabilities reports, commits messages, and security questions/tags on Stack Overflow.[2] The full list of keywords used in our approach can be found in Fig. 1. Figure 4 shows an example of security-critical code in Apache Tomcat identified automatically by our approach. We have also implemented a parser that can find all the files involved in previous security bug, vulnerability reports and pull-requests with security tags. This parser collects the CVE reports and extracts the version numbers of the systems in which the vulnerabilities are present. Then, it extracts the file names from the bug reports and issues reported for those versions.

[2] https://stackoverflow.com/.

Table 1 List of keywords used in our approach

Keywords			
id	socialsecuritynumber	undercover	path
userid	dateofbirth	crypted	signature
uuid	secret	hashed	role
password	confidential	top secret	hostname
pwd	classified	restricted	covered
username	login	hidden	lock
account	identifier	encrypt	algorithm
creditcard	unique	personal	salt
phonenumber	name	address	nonce
private	critical	cached	host
privacy	vulnerable	security	port
secure	authenticator	encoded	backdoor
credential	key	connectionString	payment
digital certificate	administrator	undercovered	transaction
biometrics	auth	code	ip-address
safe	authenticate	permission	transcoded
confidentiality	credentials	access	restricted access
sensitive	credit card number	token	sensitive information
admin	encrypted	certificate	sensitive data
access	hash	cover	card credit email
content	secret key	job	securitymanagement
secure	client id	protect	private field
user details	hidden field	security constraint	private
	auth constraint		member

Fig. 4 An example of a security-critical code fragments identified by our approach

After identifying the list of security-critical files, we used a multi-objective genetic algorithm, based on NSGA-II (Deb et al. 2002), to generate refactoring solutions that prioritize and fix the files associated with quality and security issues. The

quality objectives are based on code smells detected using a set of rules (Kessentini et al. 2011) and the potential improvements in QMOOD quality measures (Bansiya and Davis 2002) defined in Table 2. The QMOOD model is computed using 11 low-level design metrics that are also defined in Bansiya and Davis (2002). We considered code smells and QMOOD as separate objectives since developers may want to understand the impact of fixing the code smells on the quality attributes to reason about the relevance of recommended refactorings. The second objective estimates the importance of the refactored security-critical files based on a combination of textual, history and static analysis measures. The textual analysis is based on matching scores between the keywords and the source code files (e.g. names of code elements, comments). The static analysis calculates the coupling score of a class with other classes in the project: the most severe security-critical code fragments are the ones that are highly coupled (Chowdhury and Zulkernine 2011). The third measure counts the number of occurrences of the security-critical files in previous security bugs, vulnerability reports, and pull-requests with security tags to evaluate and refactor. Thus, the second objective will favor refactoring solutions targeting important security-critical files.

In the next sub-sections, we give details about each of these two major components.

3.1 Security-critical file detection

To detect security-critical files, we combined three different measures of textual, static and history analyses. All three measures are normalized to values between 0 and 1 using the min max formula (Yu et al. 2009). We then calculated their average score to rank the security-critical files.

With the chosen set of keywords, we calculated a textual security-critically-score for each file to estimate the extent to which the file is related to security concerns and hence needs to be protected. The higher the score is the more likely the file is security-critical. We compute the textual security-criticality-score based on cosine similarity between each file and the set of keywords. Let n be the number of files in the source code and W an array containing the set of keywords. After pre-processing the source code including tokenization, lemmatization, stop words filtering and punctuation removal, we calculate the tf-idf score considering the file $f_i, i \in \{1, 2, \ldots, n, \}$ and W as corpus. Cosine similarity is then calculated as follows:

$$sim(f_i, W) = tf - idf(f_i, W) * transpose(tf - idf(f_i, W)) \quad (1)$$

A file with security-critical code may spread its vulnerability to its connected files in the system. Therefore, we parse the source code and compute a coupling metric as a second measure for all classes in each file. The coupling metric for each file is then equal to the average of the coupling metrics of all of its classes.

We define coupling of a class as the number of Call-Ins and Call-Outs from that class (Mkaouer et al. 2016). Let m be the number of classes in file f_i, $C = \{c_1, c_2, \ldots, c_m\}$ the set of classes in f_i and cp_j the coupling metric for class c_j, the coupling metric for file f_i is:

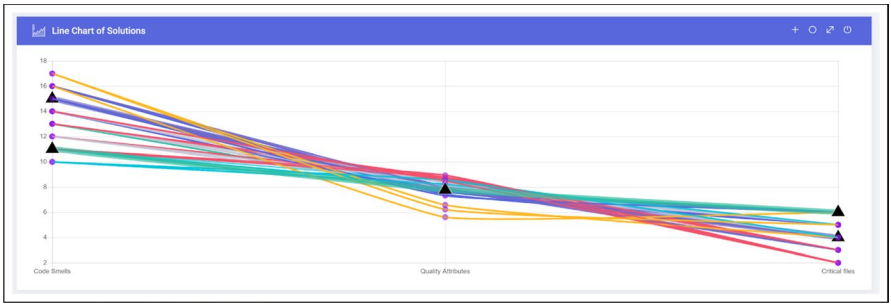

Fig. 5 An example of a Pareto front of refactoring solutions generated by our tool for OpenCSV project

$$cp_i = \frac{\sum_{j=1}^{m} cp_j}{m} \tag{2}$$

which is the average of the coupling of the classes contained in file f_i.

The third history-based measure simply counts the number of occurrences of the source code files in previous security bugs, vulnerability reports and pull-requests of code reviews with security tags. We normalized the values of the coupling metric into [0,1] using normalization min max formula. Thus, if a file with high cosine similarity score is highly coupled and appeared in previous security bugs or vulnerability reports, then it is considered as *critical* and should be refactored if the confidential data it contains is accessible or could be compromised. If a file with high cosine similarity score is not highly coupled and was not vulnerable before, then it is not urgent to be refactored. The average score of all the three measures reflect these intuitions.

3.2 Refactoring prioritization for identified security-critical code

We adapted a multi-objective search algorithm, based on NSGA-II (Deb et al. 2002), to optimize three objectives that take into account both the security and the quality of the software system. We chose this algorithm because it was used before in similar software engineering problems (Mkaouer et al. 2014b; Ouni et al. 2013a, b) and was proven to be able to balance independent or even conflicting objectives.

The algorithm is executed to find a set of non-dominated solutions balancing the objectives of security, code smells and quality attributes.With our multi-objective tool, the developer does not have to assign weights to the objectives. The user can select a solution from the Pareto-front of non-dominated solutions based on his needs and priorities as shown in Fig. 5. The x-axis represents the different objectives (i.e. security, code smells, and quality attributes). The lines are the Pareto optimal refactoring solutions (i.e those that are not dominated by any other feasible solutions). This plot gives full information on objective values and on objective trade-offs, which inform how improving one objective is related to deteriorating the second one. The developer takes this information into account while specifying the

preferred Pareto optimal objective point. The developer can also interact with our tool and give feedback by accepting or rejecting the refactoring recommendations.

In the following subsections, we summarize the adaptation of the multi-objective algorithm to our problem.

3.2.1 Solution representation

In our adaptation, a refactoring solution is an ordered vector of refactoring operations. A refactoring operation is composed of a type as well as a set of controlling parameters such as attributes, source class, target class, and methods. We used 14 different refactoring types, as follows: Encapsulate Field, Increase Field Security, Decrease Field Security, Pull Up Field, Push Down Field, Move Field, Increase Method Security, Decrease Method Security Pull Up Method, Push Down Method, Move Method, Extract Class, Extract Super class, and Extract Subclass. For every refactoring, we specified the pre- and post-conditions (detailed in Opdyke 1992) to ensure the feasibility of the operation and to ensure that we did not alter the external behavior of the code. To better understand the concept of pre- and post-conditions, let us consider the refactoring rename method *M*. Before we apply the refactoring we need to make sure that there exist no method with the same name as *M* in that class. Post-conditions would be: If *M* is overridden in any sub class, its name should be changed such that it is same as its inherited method's name. Also, all calls with the new function name in all the clients should be replaced (if any). It is important to note that the order of the refactorings in the solution vector matters. We apply refactorings in the same order as they appear in the prioritization vector. Figure 6

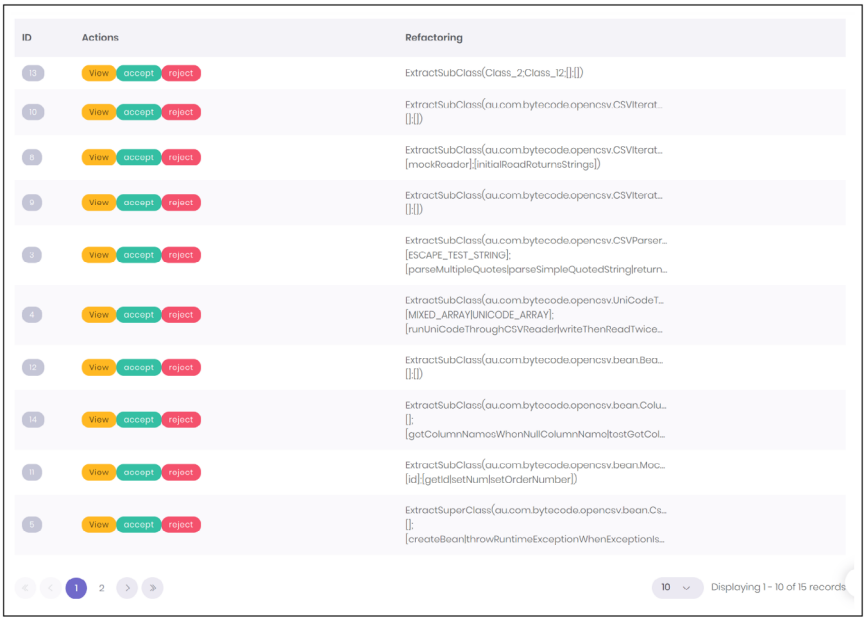

Fig. 6 An example of a solution generated by our tool

Table 2 Quality attributes and their equations (Bansiya and Davis 2002)

Quality attributes	Definition computation
Reusability	A design with low coupling and high cohesion is easily reused by other designs
	$0.25 * Coupling + 0.25 * Cohesion + 0.5 * Messaging + 0.5 * DesignSize$
Flexibility	The degree of allowance of changes in the design
	$0.25 * Encapsulation - 0.25 * Coupling + 0.5 * Composition + 0.5 * Polymorphism$
Understandability	The degree of understanding and the easiness of learning the design implementation details
	$0.33 * Abstraction + 0.33 * Encapsulation - 0.33 * Coupling + 0.33 * Cohesion - 0.33 * Polymorphism - 0.33 * Complexity - 0.33 * DesignSize$
Functionality	Classes with given functions that are publicly stated in interfaces to be used by others
	$0.12 * Cohesion + 0.22 * Polymorphism + 0.22 * Messaging + 0.22 * DesignSize + 0.22 * Hierarchies$
Extendibility	Measurement of a design's ability to incorporate new functional requirements
	$0.5 * Abstraction - 0.5 * Coupling + 0.5 * Inheritance + 0.5 * Polymorphism$
Effectiveness	Design efficiency in fulfilling the required functionality
	$0.2 * Abstraction + 0.2 * Encapsulation + 0.2 * Composition + 0.2 * Inheritance + 0.2 * Polymorphism$

shows an example of a solution generated by our multi-objective tool containing q sequence of refactorings for OpenCSV and the user can interact with: by accepting, modifying and rejecting any of them.

3.2.2 Quality metrics

Bansiya and Davis (2002) provided a set of concrete measures of software quality by defining a statistical model for object-oriented programs. This model, called Quality Model for Object Oriented Design (QMOOD), is composed of six high-level design quality attribute metrics (defined in Table 2) and these are calculated using 11 lower-level metrics. We selected this model because it is commonly used in industry to estimate code quality (O'Keeffe and Cinnéide 2008; Jensen et al. 2010; Lee et al. 2011), and in refactoring studies (Alizadeh and Kessentini 2018; Harman and Tratt 2007; Ouni et al. 2013a; Shatnawi and Li 2011).

3.2.3 Fitness functions

Our approach takes into consideration 3 objectives: the first one is the sum of the relative changes of the 6 QMOOD attributes after applying a refactoring solution. This objective can be written as follows:

$$\sum_{n=1}^{6} \frac{Q_i^{after} - Q_i^{before}}{Q_i^{before}} \tag{3}$$

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

Most code anti-patterns can be detected using interface and code quality metrics. In our study, we used an existing antipattern detection tool based on rules (Kessentini et al. 2011) that can detect 11 types of antipatterns defined in Table 3. The description of these antipatterns can be found in the website associated with this paper (CWE 2009). We have chosen this tool because of its high accuracy. Using this measure we have defined the second fitness function as the value of the anti-patterns "fixed" by the refactoring solution. This objective can be written as follow:

$$\sum_{i=1}^{FS} antipatterns_i \tag{4}$$

where FS is the total number of files in the system and $antipatterns_i$ is the number of fixed antipatterns in the file i by the refactorings solution.

In the third fitness function, we maximize the number of critical files to refactor. This objective can be written as follows:

$$\sum_{i=1}^{F} Severity_i \tag{5}$$

Table 3 List of the detected antipatterns (Brown et al. 1998; Fowler 2018; Palomba et al. 2014)

Antipattern	Definition
BLOB	A class that is too large and not cohesive enough, that monopolises most of the processing, takes most of the decisions, and is associated to data classes
Lazy class	A class that has few fields and methods (with little complexity)
Long method	A class that has a method that is overly long, in term of LOCs
Long parameter list	A class that has (at least) one method with a too long list of parameters with respect to the average number of parameters per methods in the system
Message chains	A class that has (at least) one method with a too long list of parameters with respect to the average number of parameters per methods in the system
Refused parent bequest (RPB)	A class that redefines inherited method using empty bodies, thus breaking polymorphism
Spaghetti code	A class declaring long methods with no parameters and using global variables. These methods interact too much using complex decision algorithms. This class does not exploit and prevents the use of polymorphism and inheritance
Feature envy	A method making too many calls to methods ofanother class to obtain data and/or functionality
Inappropriate intimacy	Occurs when one class accesses to the internal parts that should be private of another class
CDSBP	A class that exposes its fields, thus violating the principle of encapsulation
Speculative generality	A class that is defined as abstract but that has very few children, which do not make use of its methods

where F is the total number of selected critical files and $severity_i$ is the severity score of file i selected for refactoring. This severity score is the average of the three textual, history and static measures described previously.

4 Experiment and results

In this section, we first present our research questions and validation methodology followed by our experimental setup and our results.

4.1 Research questions

We defined three main research questions to measure the relevance, efficiency and usefulness of our approach comparing to the state of the art (Ouni et al. 2016; Fokaefs et al. 2011) based on several practical scenarios. It is important to evaluate, first, the manual correctness of the recommended refactorings. Since it is not sufficient to make correct recommendations, we evaluated the efficiency of these refactorings by ranking their importance to developers. In practice, they are not interested

to check and apply *all* the correct refactorings due to limited resources but they focus on the most important ones before the release deadline. We have also used post-study questionnaires to evaluate the benefits of our approach and the relevance of our results.

The three research questions are as follows:

RQ1. *Relevance and comparison to existing refactoring techniques* To what extent are the refactorings recommended by our approach relevant, compared to existing refactoring techniques based on improving quality measures (Ouni et al. 2016; Fokaefs et al. 2011)?

RQ2. *Ranking efficiency* To what extent can our approach *efficiently rank* recommended refactorings compared to existing techniques (Ouni et al. 2016; Fokaefs et al. 2011)?

RQ3. *Insights* How do programmers evaluate the impact of our approach in practice?

To answer *RQ1*, we validated our approach on six medium to large-size open-source systems and one industrial project to manually evaluate the relevance of the recommended refactorings based on both quality and security. The relevance of recommended refactorings corresponds to the manual inspection by developers if they found the refactoring correct to apply. In that case, the developers chose the action to 'accept' a refactoring recommendation as shown in Fig. 6.

To this end, we used the *Manual Correctness (MC@k)* precision metric. MC@k denotes the number of correct refactorings in the top k recommended refactorings by the solution divided by k. It is unrealistic to calculate the recall since it requires the inspection of the entire system. We further address *RQ1* by interviewing the participants who analyzed the output of our approach on the industrial project, who are among the original developers of that system (as detailed in the next section).

We asked a group of 32 participants to manually evaluate the relevance of the top k refactorings that they selected using the different tools. We compared our approach to two fully-automated refactoring tools: Ouni et al. (2016) and JDeodorant (Fokaefs et al. 2011). Ouni et al. (2016) proposed a multi-objective refactoring formulation based on NSGA-II that generates a solution to maximize treatment of several quality attributes and antipatterns. JDeodorant (Fokaefs et al. 2011) is an Eclipse plugin to detect antipatterns and recommended refactorings based on a set of templates. As JDeodorant supports a lower number of refactoring types with respect to the ones considered by our tool, we restrict our comparison with it to just these refactorings. We compared our work to JDeodorant and Ouni et al. (2016) because (1) they are the only publicly available automated/semi-automated JAVA refactoring recommendation tools (most of existing tools are more related to detection of refactoring opportunities and not semi-automated/automated refactoring recommendations) and the used techniques are different (deterministic versus search-based); (2) both studies are focusing on improving similar quality attributes of our work but without considering the security aspect; and (3) they were evaluated in previous studies based on the same systems used in this paper. We believe that this benchmark is adequate to show the value of

Table 4 Demographics of the studied projects

System	Release	# Classes	KLOC	GitHub link
tink	v1.2.2	590	185	google/tink.git
pac4j	v3.6.1	975	67	pac4j/pac4j.git
atomix	v3.0.11	2719	188	atomix/atomix.git
securitybuilder	v1.0.0	313	81	tersesystems/securitybuilder.git
rest.li	v15.0.3	4185	478	linkedin/rest.li.git
firefly	v4.9.5	2188	154	hypercube1024/firefly.git
DAS	v7.6.1	973	326	N.A.

considering security critical files at the top of typical quality metrics improvement (used in those existing tools). The comparison with Ouni et al. (2016) can confirm that the outperformance compared to JDeodorant is not due to the search algorithm itself.

Furthermore, we implemented a sanity check approach where we used our multi-objective algorithm with only the quality and antipatterns objectives and then ranked the recommended refactorings based on the security severity measure. Thus, we can evaluate the benefits of considering maximizing the refactoring of security-critical files with quality issues as a separate objective rather than using that function to rank the recommended refactorings. Finally, we compared our work with a mono-objective genetic algorithm combining all the three objectives into one function with equal weights so we can evaluate whether the various objectives are conflicting.

We note that the mono-objective approach and JDeodorant only provide one refactoring solution while the other algorithms generate sets of non-dominated solutions. To make meaningful comparisons, we selected the best solution for the multi-objective algorithms using a knee-point strategy. The knee point corresponds to the solution with the maximal trade-off between the objectives. Thus, we selected the knee point from the Pareto approximation having the median hyper-volume *IHV* value. By that strategy, we ensure fairness when making comparisons against the mono-objective and deterministic techniques.

The antipatterns and internal quality indicators were not used as proxies for estimating the refactoring relevance since the developers' manual evaluation already includes a review of the impact of suggested changes on quality. We also wanted to avoid any bias in our experiments since antipatterns and quality attributes are considered in the fitness functions of our approach. Furthermore, not all the refactorings that improve a quality attribute are relevant to the developers. The only fair way to evaluate the relevance of our tool is thus manual evaluation of the results by active developers.

To answer *RQ2*, we evaluated the ranking efficiency of the refactorings by asking the participants to manually rate their importance: high, medium, or low. Then, the evaluation metric *importance@k* calculates the number of refactorings rated "high" in the top k, divided by k. Of course, this measure is applied in the order of refactorings generated by the various approaches.

Table 5 Selected programmers

System	# subjects	Avg. prog. exp.	Avg. refactoring exp.
tink	5	6.5	Very high
pac4j	5	7.5	High
atomix	5	9	High
securitybuilder	5	8	Very high
rest.li	5	8	Very high
firefly	5	9	High
DAS	2	12.5	Very high

To answer *RQ3*, we used a post-study questionnaire that collected the opinions of developers on our tool and the relevance of refactoring security-critical files on software projects.

4.2 Software projects and experimental setting

4.2.1 Studied projects

We used a set of well-known open-source and one system from our industrial partner, a software company with a focus on e-commerce and web development. We applied our approach to six open-source Java projects: tink, pac4j, atomix, securitybuilder, rest.li and firefly. Tink provides a simple and misuse-proof API for common cryptographic tasks. Pac4j is a security engine system. Atomix is an event-driven framework for coordinating fault-tolerant distributed systems. Securitybuilder is a fluent builder API for JCA and JSSE classes and especially X.509 certificates. Rest.li is a framework for building scalable RESTful architectures. Firefly is an asynchronous framework for rapid development of high-performance web application. Among the 6 systems, there are only 2 security projects that we selected intentionally to check if our approach can propose similar results to non-security projects.

To get feedback from the original developers of a system, we ran our experiment on a large industrial project, called DAS, provided by our industrial partner. The analyzed project can collect, analyze and synthesize a variety of data and sources related to online customers such as their shopping behavior. It was implemented over a period of 9 years, frequently changed over time, and had experienced several vulnerabilities.

We selected these systems for our validation because they range from medium to large-sized and have been actively developed over several years, they are widely used by companies as third party code and several previous vulnerabilities were detected on them. Table 4 provides some demographic data on these systems. The data collected on these systems included the history of bug reports, vulnerability

reports and pull-requests to identify ones with security tags. The security bugs and vulnerabilities were extracted from The Common Vulnerabilities and Exposures database (CVE) (Cve 2021). We also downloaded bug reports from Bugzilla;[3] then we filtered them using the list of security keywords that we used as well to detect security critical files (Fig. 1). The list of keywords [defined in the literature (Walden et al. 2014; Scandariato et al. 2014; Tang et al. 2015)] are the most common security related words that developers may use in naming code elements, writing comments, security bugs and vulnerabilities reports, commits messages, and security questions or tags on Stack Overflow.[4]

4.2.2 Subjects

Our study involved 30 graduate students and 2 software developers from the industrial partner. Participants included 24 Master's students in Software Engineering, 6 Ph.D. students in Software Engineering and 2 software developers. All participants were volunteers who were familiar with software security, refactoring, Java and quality assurance. All the Master's students were working full-time in industry as developers, managers, or architects. They average 6 years of experience in industry and 16 out of the 24 have worked on either fixing security bugs or patching vulnerabilities. Participants were first asked to fill out a pre-study questionnaire containing six questions. The questionnaire helped to collect background information such as their role within the company, their programming experience, and their familiarity with software refactoring and security. Although the vast majority of participants were already familiar with refactoring, all the participants attended one lecture of 2 h on software refactoring by the organizers of the experiments. The details of the selected participants can be found in Table 5, including their programming experience (years) and level of familiarity with refactoring. Each participant was asked to assess the meaningfulness of the refactorings recommended after using one of the five tools on one system to avoid a training threat. The participants did not only evaluate the suggested refactorings but were asked to configure, run and interact with the tools on the different systems. The only exceptions were related to the two participants from the industrial partner where they agreed to evaluate only their industrial software. We assigned tasks to participants according to the studied systems, the techniques to be tested and developers' experience. To avoid any potential bias, we did not share with the participants the goal of the study to validate the hypothesis about if security metrics are important to consider, beyond just quality metrics, when refactoring the source code. To ensure a fair comparison, none of the participants used any of the studied tools before.

[3] https://www.bugzilla.org/.
[4] https://stackoverflow.com/.

4.2.3 Experimental setting

For each algorithm and for each system, we performed a set of experiments using several population sizes: 50, 100, 150 and 200. Then, we specified the maximum chromosome length (maximum number of refactorings). The resulting vector length is proportional to the size of the program to refactor. Thus, the upper and lower bounds on the chromosome length were set to 10 and 100, respectively. The stopping criterion was set to 10,000 fitness evaluations for all algorithms to ensure fairness. To have significant results, for each pair (algorithm, system), we used a trial and error method (Arcuri and Briand 2011) for parameter configuration. Trial and error is a fundamental method of problem solving. It is characterized by repeated and varied attempts of algorithm configurations.

4.3 Results

Results for RQ1 Figures 7 and 8 confirm that our approach was able to identify relevant refactorings among the top recommendations on the six open source systems and the industrial project. Figure 7 shows the average manual correctness (MC@k) results of our technique on the different seven systems, with k ranging from 3 to 10. For example, all the recommended refactorings in the top 3 are considered relevant by the participants. Most of the refactorings recommended by our Multi-Objective refactoring search (MORS) approach in the top 5 (k = 5) are relevant with an average MC@5 of 92%. The lowest average of manual correctness is 70% for k = 10 which is still acceptable since it means that just 3 recommendations out of the 10 are not relevant (even if they are correct). For instance, the refactoring recommendations on the DAS (industrial) system, evaluated by the original developers, are considered all correct for k = 3 and k = 5 and only two refactorings were not relevant

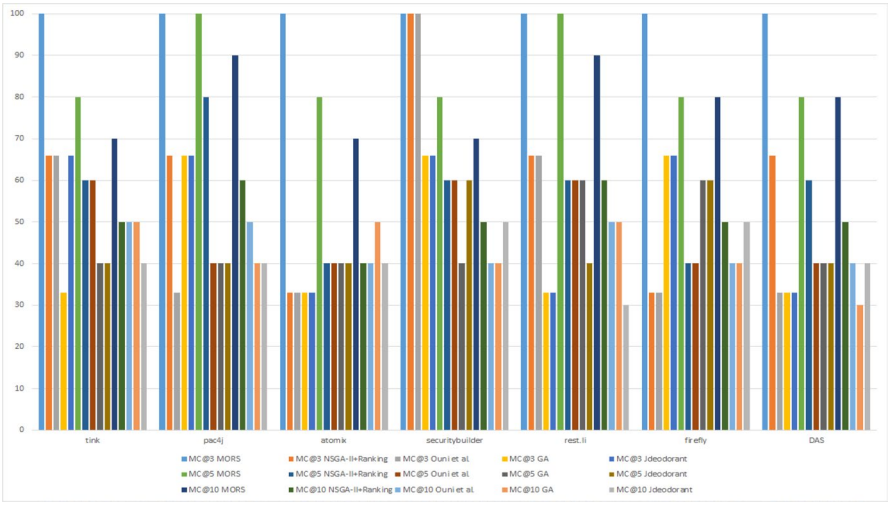

Fig. 7 The manual evaluation scores (MC@k) on the seven systems with k = 3, 5 and 10

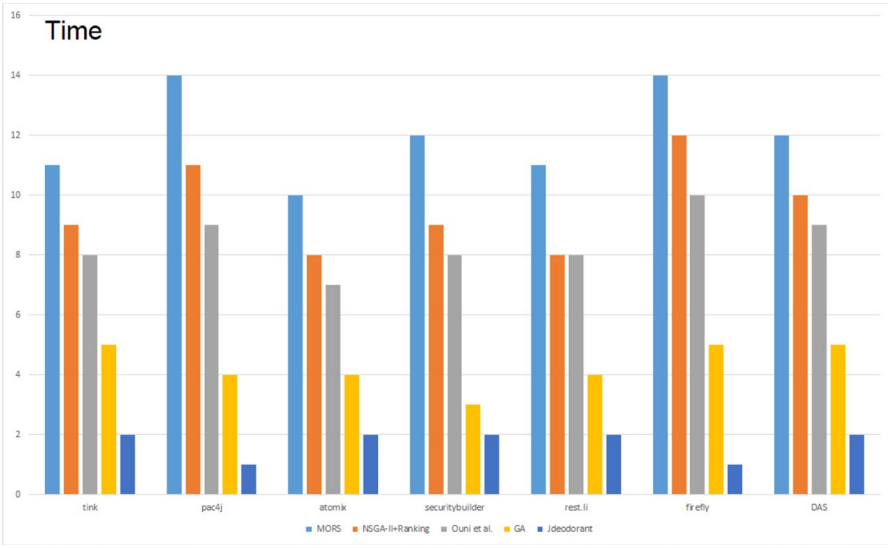

Fig. 8 Average execution time, in minutes, on the seven systems

for k = 10. It is normal that some refactorings related to security-critical files may not be considered relevant for several reasons such as the risk.

to break the code versus their benefits. However, our results suggest that developers are interested to refactor security-critical files, particularly considering that the participants were not aware of the goals of the study (e.g., security). Figure 8 summarizes the execution times of our approach on the different systems. The average execution time is 11 min. The highest execution time was observed on the industrial system (14 min). It is normal that execution time is correlated with the size of the analyzed systems since the tool has to parse the files to identify the most security-critical ones, and then run NSGA-II with the different three objectives. We consider the execution times reasonable since we are not addressing a real-time problem. Furthermore, execution times can be reduced further during subsequent executions of the tool since we may only focus on recently modified, instead of running the algorithms on the entire system.

In terms of comparison with existing refactoring techniques, it is clear from the results that our Multi-Objective refactoring search (MORS) approach generated more relevant refactorings as compared to the tools of Ouni et al., JDeodorant, the mono-objective search, and a multi-objective search based on two quality objectives combined with a ranking of refactorings based on the security measure (NSGA-II+Ranking). When manually comparing the results of the different tools, we found that the remaining automated refactorings generated a lot of refactorings comparing to our approach. In fact, the participants were not interested to blindly change anything in the code just to improve quality attribute measures. The mono-objective search performed worse than the different multi-objective approaches on all the systems which confirms the conflicting nature of the different objectives. In other words, the aggregation of the quality and security objectives in a mono-objective

algorithm reduces the performance of the search algorithm compared to multi-objective algorithms where they are treated as separate objectives. The reason is that improving the quality metrics may deteriorate the security metrics of the system. For example, extracting subclasses from a superclass, may expand the attack surface if the superclass contains security-critical attributes and methods. This is due to the fact that if an attacker can access the superclass and inject code then this security thread can be expanded easily and quickly to all its children classes. Thus, our hypothesis is that quality and security are two conflicting objectives which implies that the mono-objective search is not appropriate to solve this problem.

In terms of execution time, we found that our performance was worse than existing techniques, due to the higher number of objectives and parsing the files to identify the critical ones. JDeodorant has the best execution time of approximately 2 min per project. To conclude, the execution time is still reasonable for all the studied approaches, especially considering that there are no hard time constraints when performing refactoring.

To answer *RQ1*, our results for the six open source systems and the industrial system using the different evaluation metrics of relevance and execution time clearly validate the hypothesis that our approach can generate relevant refactorings for security-critical files.

Results for RQ2 To evaluate the efficiency of our approach in ranking the refactoring operations based on the combination of quality and security objectives, we used the importance@k measure to evaluate the importance of recommended refactorings. The participants only considered refactorings that were evaluated as relevant in *RQ1*. Thus, the goal is to validate the hypothesis that the refactorings related to security-critical files are the most important ones, from the developers' perspectives, by comparing the proposed ranking to existing approaches that ranking primarily on code quality measures.

Figure 9 indicates that the use of the three objectives of code quality and security to find and rank the refactorings based on NSGA-II leads to efficient rankings of recommended refactorings. The majority of the identified refactorings located in the top 3, 5 and 10 were rated high in terms of importance by the participants. An average of 100%, 91%, and 83% of importance@k scores are achieved for k = 3, 5, and 10 respectively on all the systems. It is clear from the figure as well that our MORS approach outperforms all the other techniques including the ranking of the refactorings based on the security measure after running NSGA-II on only the two quality objectives (NSGA-II + Ranking). This confirms the relevance of our choice to integrate security as a separate objective to help the algorithm converge on refactorings targeting security-critical files. However, the NSGA-II + Ranking approach outperformed all the existing refactoring approaches based only on quality measures to rank the recommended refactorings.

Results for RQ3 We summarize in the following the feedback of the developers based on the post-study questionnaire.

All participants agreed on the benefits of refactoring security-critical code. They mentioned a number of advantages such as the early identification and prevention of vulnerabilities, the reduction of security breaches as well as the maintenance effort and prioritizing of files that should be carefully reviewed and

Table 6 Top 3 ranked solutions by the participants

Solution ID	# of refactored security-critical files	Sum of quality metrics	Sum of code smells	Ranking
1164622	10	0.83367444	21	1
1164645	5	1.04797722	12	3
1164651	15	0.67012856	4	2

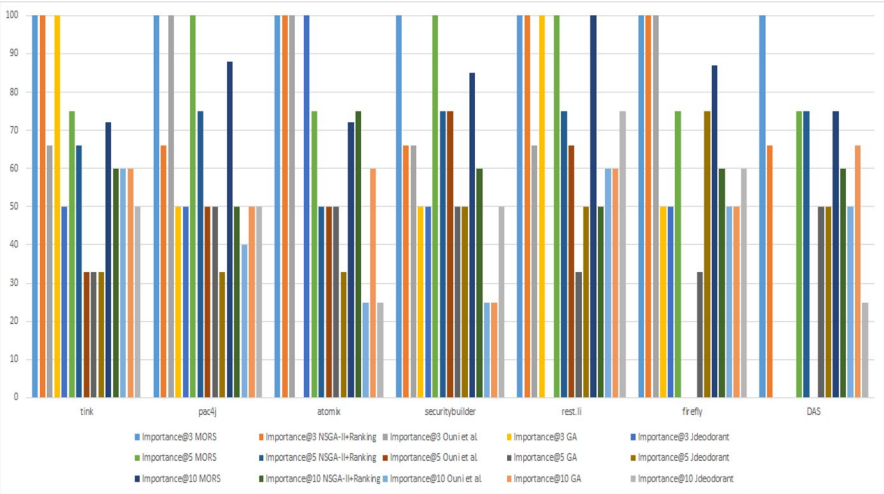

Fig. 9 The importance@k scores on the seven systems with k = 3, 5 and 10

refactored before approving new commits or releases. Some participants highlighted the benefit of catching test credentials that developers forgot to remove via some refactorings, which is a common mistake among developers. The misuse of these hard-coded credentials is actually an example of the Broken Authentication vulnerability, which is ranked second among the OWASP-TOP 10 Vulnerabilities in web applications in 2018.

The participants emphasized the relevance of refactoring security-critical code fragments for the potential increase in code quality and avoiding confidential data exposure which may result in less cost and better reputation for the organization. Two developers also mentioned the benefit of predicting and avoiding security issues when integrating third party code. For instance several companies are concerned about vulnerabilities when integrating open source projects. Table 6 shows three different refactoring solutions, their fitness values and the ranking assigned by the participants. Solution 1164622 improves quality and security each moderately. Solution 1164645 focuses on improving quality metrics and fixing code smells more than the security aspect of the system. Finally, solution 1164651 is more concerned about refactoring security-critical files than improving the quality of the code. The results reveal that most participants prefer solution 1164622 that finds

a good trade-off between the quality and security of the system. Solution 1164651 and 1164645 are ranked second and third, respectively, which shows that most of the participants prioritized security as the most critical reason for refactoring, even compared to improving quality metrics which is the second most important motivation for refactoring.

Several comments mentioned the potential use of our tool at different stages of the software life cycle, whether during the development stage where developers are alarmed that they are dealing with critical-code, or during code reviews before releases when reviewers focus on the changes made on that portion of code, or even during documentation where all these alerts are recorded for developers in the future. The participants see our tool as relevant for existing continuous integration (CI) and continuous delivery (CD) tools.

Finally, all developers confirmed that they have never used or heard of a tool that leverages this technique for automated refactoring of security-critical code. The practitioners from the industrial partner confirmed that they are not aware of a similar tool. Currently security-critical code fragments are manually identified and refactored during code reviews and a lot of them are missed during that process. And the proposed tool has subsequently been licensed to this partner (in collaboration with the university technology transfer office of the authors).

5 Threats to validity

In our experiments, construct validity threats are related to the absence of similar work that prioritize refactoring for both security and quality purposes. For that reason, we compared our proposal mainly with existing studies that focus on improving quality via refactoring. A construct threat can also be related to the corpus of data used in our experiments since it may introduce some noise to the quality of our results especially with the subjective nature of refactoring. Since we used a variety of computational search and machine learning algorithms, the parameter tuning used in our experiments creates an internal threat that we need to evaluate in our future work. The parameter values used in our experiments are found by trial-and-error. However, it would be an interesting perspective to design an adaptive parameter tuning strategy for our approach so that parameters are updated during the execution in order to provide the best possible performance.

The variation of correctness and speed between the different participants when using our approach and other tools can be one internal threat. Our approach may not be the only reason for the superior performance because the participants have different programming skills and familiarity with refactoring tools. To counteract this, we assigned the developers to different systems according to their programming experience so as to reduce the gap between the different groups, and we also adopted a counter-balanced design. Regarding the selected participants, we have taken precautions to ensure that our participants represent a diverse set of software developers with experience in refactoring, and also that the groups formed had, in some sense, a similar average skill set in the refactoring area.

Also, the fact that we did not ask all the participants to evaluate all the systems using all the tools can be considered another threat to the validity of our work. The reason is that it is not reasonable to ask programmers to evaluate more than 30 executions per algorithm to perform the statistical tests. We sacrificed a bit the rigor of the analysis to have our research validated by practitioners and get meaningful results.

External validity refers to the generalization of our findings. In this study, we performed our experiments on 6 different widely-used open-source systems belonging to the different domains and with different sizes and one industrial project. We considered a mix of security and non-security projects to evaluate the performance of our approach. However, we cannot assert that our results can be generalized to other applications, to programming languages other than JAVA, and to other developers than the 32 participants of our experiments.

6 Related work

6.1 Code fragments accessibility

Grothoff et al. (2007) present a tool called JAMIT to restrict access modifiers from security perspective. Specifically, the authors analyzed whether a class is confined to the package to which it is declared so the goal is to guarantee that a reference to a class cannot be obtained outside its package. The validation focused on reporting the percentage of classes that could be confinable.

Bouillon et al. (2008) present a tool that checks for over-exposed methods in Java applications. Their tool determines the best access modifier by analyzing the references to each method. Müller (2010) uses bytecode analysis to detect those access modifiers of methods and fields that should be more restrictive.

Steimann and Thies (2009) highlight the difficulties of carrying out refactoring in the presence of non-public classes and methods. The authors formalize accessibility constraints in order to check the preconditions of a refactoring (e.g., moving a class to another package requires checking whether the accessibility of the class allows its users to still reference it). In particular, the authors analyze the cases in which a class or a method is moved between packages or classes with the goal of adapting their access modifiers to preserve the original behavior.

Zoller and Schmolitzky (2012) present a tool called AccessAnalysis to detect over-exposed methods and classes by analyzing the references to code elements. Kobori et al. (2015) investigated the evolution of over-exposed methods and fields for a set of open-source applications. They reported that the change of access modifiers of methods is not frequent. They also found that the number of over-exposed methods and fields tends to increase in time.

Vidal et al. (2016b), presented two empirical studies on over-exposed methods with the goal of analyzing their impact on information hiding and the interfaces of classes, and over exposed classes (Vidal et al. 2016c). In both studies, they analyzed the history of the systems with the goal of understanding the variations in the over-exposed methods. They expanded and improved their work on method accessibility

to class accessibility in Vidal et al. (2016c) and presented an Eclipse plugin to make component public interfaces match with the developer's intent.

To summarize, the goal of this first category of work is mainly to use static analysis to identify over-exposed code fragments whether related to security or not but without recommending refactorings.

6.2 Software security metrics

In this category of studies, the main focus is to measure the security of software components (Agrawal and Khan 2012, 2014; Alshammari et al. 2009, 2010a; Wang et al. 2018; Wright et al. 2013; Srivastava and Kumar 2018; Chowdhury et al. 2008).

Chowdhury et al. (2008), proposed an approach to measure the security of the code using a set of quality metrics. They proposed metrics that aim to assess how securely a system's source code is structured. The metrics are stall ratio, coupling corruption propagation, and critical element ratio. One shortcoming related to this work is that some of the metric values are decided based on intuition. For example, finding out the critical elements in a class depends on the intuition of the data collectors and it should be manually tagged.

Alshammari et al. (2010a), presented a set of metrics to measure the security of each class in an object oriented design projects. To measure the security of each class, they utilized two properties of object oriented design: the accessibility of, and interactions within, classes. To measure the security of object oriented design, they defined the metrics based on quality metric, including composition, coupling, extensibility, inheritance, and design size. In order to identify if an attribute is critical (i.e. carrying critical information), they assumed that developers/designers have annotated class diagrams such as UMLsec and SPARK's annotations with a secrecy tag for each critical attribute in the design.

Agrawal and Khan (2012) presented an investigation of how coupling induces vulnerability propagation in an object oriented design. They introduce a metric to measure Coupling Induced Vulnerability Propagation Factor (CIVPF) for an object oriented design. Their main idea behind this research is that Coupling is one of the means responsible for the vulnerability propagation. In order to compute CIVPF, they introduce some characteristics for an attribute to be vulnerable. Then, they defined the vulnerable method and class based on their access to vulnerable attributes.

The same authors later studied the role of cohesion for object oriented design security and proposed security metrics measuring the impact of cohesion on security vulnerability (Agrawal and Khan 2014). Highly cohesive classes are more understandable, modifiable and maintainable (Agrawal and Khan 2014). Their work is based on the assumption that in object oriented design, when a vulnerable attribute is spreading from one class to another it may compromise the whole system. They have proposed three metrics to measure the vulnerable association of a method in a vulnerable class, vulnerable association within a class and vulnerable association of an object oriented design. However they claim that computing these three metrics does not require any type of documents including Collaboration Diagrams,

Sequence Diagram, State Diagram and Class Hierarchy, but it does require that an attribute should be labeled as vulnerable manually.

6.3 Refactoring for security

Maruyama and Omori (2011) presented a tool named Jsart (Java security-aware refactoring tool) that supports two types of refactorings related to software security, which is built as an Eclipse plug-in. It helps programmers to estimate the impact of the application of refactorings on security characteristics of the changed files by detecting the downgrading of the access level of a field variable within the modified code. In another study, the authors in Alshammari et al. (2010b) and Mumtaz et al. (2018) assessed the impact of refactoring rules on the security of object-oriented applications by computing security metrics and code smells before and after refactoring. However, the two studies mentioned above do not provide refactoring recommendations to developers to help them know what refactorings to use and what classes they should target.

Ghaith et al. (2012) present a search-based approach to automate the refactoring process while improving software security. They used the search-based refactoring platform, Code-Imp, to refactor the code. The fitness function used to guide the search is based on a set of software security metrics they collected from existing work. However, the main objective of the refactoring process is to improve the security of the system and they did not focus on the quality of the code and design.

6.4 Search-based refactoring

Search-based techniques (Harman and Jones 2001; Kessentini et al. 2010; Mansoor et al. 2017) are widely studied to automate software refactoring where the goal is to improve the design quality of a system based mainly on a set of software metrics. The majority of existing work combines several metrics in a single fitness function to find the best sequence of refactorings. Seng et al. (2006) have proposed a single-objective optimization approach using a genetic algorithm to suggest a list of refactorings to improve software quality. The work of O'Keeffe and Cinnéide (2008) uses various local search-based techniques such as hill climbing and simulated annealing to provide an automated refactoring support. They use the QMOOD metrics suite to evaluate the improvement in quality. The majority of existing multi-objective refactoring techniques (Harman and Tratt 2007; Ouni et al. 2016; Mkaouer et al. 2015; Cinnéide et al. 2012; Kessentini et al. 2018) propose as output a set of non-dominated refactoring solutions (the Pareto front) that find a good trade-off between the considered maintainability objectives. This leaves it to the software developers to select the best solution from a set of possible refactoring solutions, which can be a challenging task as it is not natural for developers to express their preferences in terms of a fitness function value. Thus, the exploration of the Pareto front is still performed manually.

Some recent studies (Lin et al. 2016; Alizadeh et al. 2018) extended a previous work (Mkaouer et al. 2014b) to propose an interactive search based approach for refactoring recommendations. The developers have to specify a desired design at the architecture

level then the proposed approach try to find the relevant refactorings that can generate a similar design to the expected one. In our work, we do not consider the use of a desired design, thus developers are not required to manually modify the current architecture of the system to get refactoring recommendations. Furthermore, developers maybe interested to change the architecture mainly when they want to introduce an extensive number of refactorings that radically change the architecture to support new features.

7 Conclusion

We have presented an approach to recommend refactorings for security critical files while concurrently improving the code quality of a software project. We used the history of vulnerabilities and security bug reports along with a selected set of keywords (Alshammari et al. 2009, 2010a) to automatically identify security-critical files in a project based on source code, bug reports, pull-request descriptions and commit messages. After identifying these security-related files we estimated their risk based on static analysis to check their coupling with other components of the project. Then, our approach recommended refactorings to prioritize fixing quality issues in these security-critical files to improve code quality measures and remove code smells using multi-objective search. We evaluated our approach on six open source projects and one industrial system to check the relevance of our refactoring recommendations. Our results confirm the effectiveness of our approach as compared to existing refactoring approaches.

We are planning, as part of our future work, to extend our validation with a larger set of systems and data sets and to study the potential correlations between security and code quality metrics during the refactoring process.

References

Abid, C., Kessentini, M., Alizadeh, V., Dhouadi, M., Kazman, R.: How does refactoring impact security when improving quality? A security-aware refactoring approach. IEEE Trans. Softw. Eng

Agrawal, A., Khan, R.: Role of coupling in vulnerability propagation. Softw. Eng. **2**(1), 60–68 (2012)

Agrawal, A., Khan, R.: Assessing impact of cohesion on security-an object oriented design perspective. Pensee **76**(2), 161–167 (2014)

Alizadeh, V., Kessentini, M.: Reducing interactive refactoring effort via clustering-based multi-objective search. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 464–474. ACM (2018)

Alizadeh, V., Kessentini, M., Mkaouer, W., Ocinneide, M., Ouni, A., Cai, Y.: An interactive and dynamic search-based approach to software refactoring recommendations. IEEE Trans. Softw. Eng. **46**(9), 932–961 (2018)

Alshammari, B., Fidge, C., Corney, D.: Security metrics for object-oriented class designs. In: 9th International Conference on Quality Software, 2009. QSIC'09, pp. 11–20. IEEE (2009)

Alshammari, B., Fidge, C., Corney, D.: Security metrics for object-oriented designs. In: Software Engineering Conference (ASWEC), 2010 21st Australian, pp. 55–64. IEEE (2010a)

Alshammari, B., Fidge, C., Corney, D.: Assessing the impact of refactoring on security-critical object-oriented designs. In: Asia Pacific Software Engineering Conference, pp. 186–195. IEEE (2010b)

Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: 2011 33rd International Conference on Software Engineering (ICSE), pp. 1–10. IEEE (2011)

Bansiya, J., Davis, C.G.: A hierarchical model for object-oriented design quality assessment. IEEE Trans. Softw. Eng. **28**(1), 4–17 (2002)

Bouillon, P., Großkinsky, E., Steimann, F.: Controlling accessibility in agile projects with the access modifier modifier. In: International Conference on Objects, Components, Models and Patterns, pp. 41–59. Springer (2008)

Brown, W.H., Malveau, R.C., McCormick, H.W., Mowbray, T.J.: AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley, Hoboken (1998)

Chowdhury, I., Zulkernine, M.: Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. J. Syst. Archit. **57**(3), 294–313 (2011)

Chowdhury, I., Chan, B., Zulkernine, M.: Security metrics for source code structures. In: Proceedings of the Fourth International Workshop on Software Engineering for Secure Systems, pp. 57–64. ACM (2008)

Cinnéide, M.Ó, Tratt, L., Harman, M., Counsell, S., Moghadam, I.H.: Experimental assessment of software metrics using automated refactoring. In: Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 49–58. ACM (2012)

Cusumano, M.A.: Who is liable for bugs and security flaws in software? Commun. ACM **47**(3), 25–27 (2004)

Cve vulnerability data. https://www.cvedetails.com/ (2021)

CWE - 398: Indicator of Poor Code Quality. https://www.cvedetails.com/cwe-details/398/Indicator-of-Poor-Code-Quality.html (2009)

Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. **6**(2), 182–197 (2002)

Fokaefs, M., Tsantalis, N., Stroulia, E., Chatzigeorgiou, A.: Jdeodorant: identification and application of extract class refactorings. In: 2011 33rd International Conference on Software Engineering (ICSE), pp. 1037–1039. IEEE (2011)

Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, Boston (2018)

Ghaith, S., Cinnéide, M.Ó.: Improving software security using search-based refactoring. In: International Symposium on Search Based Software Engineering, pp. 121–135. Springer (2012)

Grothoff, C., Palsberg, J., Vitek, J.: Encapsulating objects with confined types. ACM Trans. Program. Lang. Syst. (TOPLAS) **29**(6), 32 (2007)

Haldar, V., Chandra, D., Franz, M.: Dynamic taint propagation for java. In: Proceedings of the 21st Annual Computer Security Applications Conference, ACSAC '05, pp. 303–311. IEEE Computer Society (2005)

Han, J., Zheng, Y.: Security characterisation and integrity assurance for software components and component-based systems. In: Proceedings of 1998 Australasian Workshop on Software Architectures, Melbourne, pp. 83–89 (1998)

Harman, M., Jones, B.F.: Search-based software engineering. Inf. Softw. Technol. **43**(14), 833–839 (2001)

Harman, M., Tratt, L.: Pareto optimal search based refactoring at the design level. In: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, pp. 1106–1113. ACM (2007)

Huang, K., Zhang, J., Tan, W., Feng, Z.: Shifting to mobile: network-based empirical study of mobile vulnerability market. IEEE Trans. Serv. Comput. **13**(1), 144–157 (2016)

Jensen, A.C., Cheng, B.H.: 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, pp. 1341–1348. ACM (2010)

Kobori, K., Matsushita, M., Inoue, K.: Evolution analysis for accessibility excessiveness in java. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 83–90. IEEE (2015)

Krsul, I.V.: Software Vulnerability Analysis. Purdue University, West Lafayette (1998)

Kessentini, M., Wimmer, M., Sahraoui, H., Boukadoum, M.: Generating transformation rules from examples for behavioral models. In: Proceedings of the Second International Workshop on Behaviour Modelling: Foundation and Applications, p. 2. ACM (2010)

Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M., Ouni, A.: Design defects detection and correction by example. In: 2011 IEEE 19th International Conference on Program Comprehension, pp. 81–90. IEEE (2011)

Kessentini, W., Wimmer, M., Sahraoui, H.: Integrating the designer in-the-loop for metamodel/model co-evolution via interactive computational search. In: Proceedings of the 21th ACM/IEEE International

Conference on Model Driven Engineering Languages and Systems, MODELS '18, pp. 101–111. ACM, New York, NY, USA (2018). https://doi.org/10.1145/3239372.3239375
Lee, S., Bae, G., Chae, H.S., Bae, D.-H., Kwon, Y.R.: Automated scheduling for clone-based refactoring using a competent GA. Softw. Pract. Exp. **41**(5), 521–550 (2011)
Lin, Y., Peng, X., Cai, Y., Dig, D., Zheng, D., Zhao, W.: Interactive and guided architectural refactoring with search-based recommendation. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 535–546. ACM, (2016)
Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in java applications with static analysis. In: Proceedings of the 14th Conference on USENIX Security Symposium—Volume 14, SSYM'05, p. 18. USENIX Association (2005)
Mansoor, U., Kessentini, M., Wimmer, M., Deb, K.: Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm. Softw. Qual. J. **25**(2), 473–501 (2017)
Maruyama, K., Omori, T.: A security-aware refactoring tool for java programs. In: Proceedings of the 4th Workshop on Refactoring Tools, pp. 22–28. ACM (2011)
Mkaouer, M.W., Kessentini, M., Bechikh, S., Cinnéide, M.Ó.: A robust multi-objective approach for software refactoring under uncertainty. In: International Symposium on Search Based Software Engineering, pp. 168–183. Springer (2014a)
Mkaouer, M.W., Kessentini, M., Bechikh, S., Deb, K., Cinnéide, M.Ó: Recommendation system for software refactoring using innovization and interactive dynamic optimization. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, pp. 331–336. ACM (2014b)
Mkaouer, W., Kessentini, M., Shaout, A., Koligheu, P., Bechikh, S., Deb, K., Ouni, A.: Many-objective software remodularization using NSGA-III. ACM Trans. Softw. Eng. Methodol. (TOSEM) **24**(3), 17 (2015)
Mkaouer, M.W., Kessentini, M., Bechikh, S., Cinnéide, M.Ó., Deb, K.: On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. Empir. Softw. Eng. **21**(6), 2503–2545 (2016)
Mkaouer, M.W., Kessentini, M., Cinnéide, M.Ó., Hayashi, S., Deb, K.: A robust multi-objective approach to balance severity and importance of refactoring opportunities. Empir. Softw. Eng. **22**(2), 894–927 (2017)
Müller, A.: Bytecode analysis for checking java access modifiers. In: Work in Progress and Poster Session, 8th Int. Conf. on Principles and Practice of Programming in Java (PPPJ 2010), Vienna, Austria, pp. 1–4 (2010)
Mumtaz, H., Alshayeb, M., Mahmood, S., Niazi, M.: An empirical study to improve software security through the application of code refactoring. Inf. Softw. Technol. **96**, 112–125 (2018)
Nikiforakis, N., Invernizzi, L., Kapravelos, A., Acker, S. Van, Joosen, W., Kruegel, C., Piessens, F., Vigna, G.: You are what you include: Large-scale evaluation of remote javascript inclusions. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS '12, pp. 736–747. ACM (2012)
Nuuo cms. https://www.cvedetails.com/cve/CVE-2018-17890/ (2018)
O'Keeffe, M., Cinnéide, M.O.: Search-based refactoring for software maintenance. J. Syst. Softw. **81**(4), 502–516 (2008)
Opdyke, W.F.: Refactoring object-oriented frameworks. Ph.D. thesi, University of Illinois at Urbana-Champaign Champaign, IL, USA (1992)
Ouni, A., Kessentini, M., Sahraoui, H.: Search-based refactoring using recorded code changes. In: 2013 17th European Conference on Software Maintenance and Reengineering, pp. 221–230. IEEE (2013a)
Ouni, A., Kessentini, M., Sahraoui, H., Hamdi, M.S.: 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, pp. 1461–1468. ACM (2013b)
Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K., Hamdi, M.S.: Improving multi-objective code-smells correction using development history. J. Syst. Softw. **105**, 18–39 (2015)
Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K., Deb, K.: Multi-criteria code refactoring using search-based software engineering: an industrial case study. ACM Trans. Softw. Eng. Methodol. (TOSEM) **25**(3), 23 (2016)
Ouni, A., Kessentini, M., Cinnéide, M.Ó., Sahraoui, H., Deb, K., Inoue, K.: More: a multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells. J. Softw. Evol. Process **29**(5), e1843 (2017)
Palomba, F., Lucia, A. De, Bavota, G., Oliveto, R.: Anti-pattern detection: methods, challenges, and open issues. In: Advances in Computers, vol. 95, pp. 201–238. Elsevier (2014)

Scandariato, R., Walden, J., Hovsepyan, A., Joosen, W.: Predicting vulnerable software components via text mining. IEEE Trans. Softw. Eng. **40**(10), 993–1006 (2014)

Seng, O., Stammel, J., Burkhart, D.: Search-based determination of refactorings for improving the class structure of object-oriented systems. In: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, pp. 1909–1916. ACM (2006)

Shatnawi, R., Li, W.: An empirical assessment of refactoring impact on software quality using a hierarchical quality model. Int. J. Softw. Eng. Appl. **5**(4), 127–149 (2011)

Srivastava, A.K., Kumar, S.: An effective computational technique for taxonomic position of security vulnerability in software development. J. Comput. Sci. **25**, 388–396 (2018)

Steimann, F., Thies, A.: From public to private to absent: refactoring java programs under constrained accessibility. In: European Conference on Object-Oriented Programming, pp. 419–443. Springer (2009)

Tang, Y., Zhao, F., Yang, Y., Lu, H., Zhou, Y., Xu, B.: Predicting vulnerable components via text mining or software metrics? An effort-aware perspective. In: 2015 IEEE International Conference on Software Quality, Reliability and Security, pp. 27–36. IEEE (2015)

Tsantalis, N., Chatzigeorgiou, A.: Ranking refactoring suggestions based on historical volatility. In: 2011 15th European Conference on Software Maintenance and Reengineering, pp. 25–34. IEEE (2011)

Vidal, S.A., Marcos, C., Díaz-Pace, J.A.: An approach to prioritize code smells for refactoring. Autom. Softw. Eng. **23**(3), 501–532 (2016a)

Vidal, S.A., Bergel, A., Marcos, C., Díaz-Pace, J.A.: Understanding and addressing exhibitionism in java empirical research about method accessibility. Empir. Softw. Eng. **21**(2), 483–516 (2016b)

Vidal, S., Bergel, A., Díaz-Pace, J.A., Marcos, C.: Over-exposed classes in java: an empirical study. Comput. Lang. Syst. Struct. **46**, 1–19 (2016c)

Walden, J., Stuckman, J., Scandariato, R.: Predicting vulnerable components: software metrics vs text mining. In: IEEE 25th International Symposium on Software Reliability Engineering, pp. 23–33. IEEE (2014)

Wang, W., Mahakala, K.R., Gupta, A., Hussein, N., Wang, Y.: A linear classifier based approach for identifying security requirements in open source software development. J. Ind. Inf. Integr. **14**, 34–40 (2018)

Wright, J.L., McQueen, M., Wellman, L.: Analyses of two end-user software vulnerability exposure metrics (extended version). Inf. Secur. Tech. Rep. **17**(4), 173–184 (2013)

Yu, L., Pan, Y., Wu, Y.: Research on data normalization methods in multi-attribute evaluation. In: 2009 International Conference on Computational Intelligence and Software Engineering, pp. 1–5. IEEE (2009)

Zazworka, N., Seaman, C., Shull, F.: Prioritizing design debt investment opportunities. In: Proceedings of the 2nd Workshop on Managing Technical Debt, pp. 39–42. ACM (2011)

Zoller, C., Schmolitzky, A.: Measuring inappropriate generosity with access modifiers in java systems. In: 2012 Joint Conference of the 22nd International Workshop on Software Measurement and the 2012 Seventh International Conference on Software Process and Product Measurement, pp. 43–52. IEEE (2012)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.