Improving Software Security Using Search-Based Refactoring

Shadi Ghaith[1] and Mel Ó Cinnéide[2]

[1] IBM Dublin Software Lab, Ireland
sghaith@ie.ibm.com
[2] School of Computer Science,
University College Dublin, Ireland
mel.ocinneide@ucd.ie

Abstract. Security metrics have been proposed to assess the security of software applications based on the principles of "reduce attack surface" and "grant least privilege." While these metrics can help inform the developer in choosing designs that provide better security, they cannot on their own show exactly how to make an application more secure. Even if they could, the onerous task of updating the software to improve its security is left to the developer. In this paper we present an approach to *automated* improvement of software security based on search-based refactoring. We use the search-based refactoring platform, Code-Imp, to refactor the code in a fully-automated fashion. The fitness function used to guide the search is based on a number of software security metrics. The purpose is to improve the security of the software immediately prior to its release and deployment. To test the value of this approach we apply it to an industrial banking application that has a strong security dimension, namely *Wife*. The results show an average improvement of 27.5% in the metrics examined. A more detailed analysis reveals that 15.5% of metric improvement results in real improvement in program security, while the remaining 12% of metric improvement is attributable to hitherto undocumented weaknesses in the security metrics themselves.

1 Introduction

Software security is generally defined as the engineering of software so that continues to function correctly under malicious attack [15]. It includes matters such as ensuring that the software is free of vulnerabilities such as buffer overflow or unhandled errors, and that SQL queries are not formed with untrusted user input. In this paper we focus on software security defined as the exposure of *classified* data to the rest of the program [1]. Classified data is data whose release to an unauthorised user would constitute a breach of security. For example, a person's bank details are classified, while the address of particular bank branch would not be classified. If we reduce the extent to which classified data is exposed to the rest of the program, we can reduce the chance that a malicious attack on the deployed program will succeed in accessing classified data.

On the other hand, refactoring is defined as a process that improves software design without changing observable behavior [9]. While refactoring is normally carried out "by hand" using only the refactoring support provided by the IDE, more recent research has investigated the possibility of using a fully automated approach that relies on search-based refactoring [10,11,13,21]. In this approach, a search technique such as hill climbing, simulated annealing or a genetic algorithm is used to drive the refactoring process, which is guided by a software metric, or set of metrics, that the developer wishes to optimise.

In this paper we explore the possibility of using such an automated refactoring approach with the goal of automating the improvement of program security. In a sense, this use of automated refactoring is far more promising than using it to improve software design. Automated refactoring is liable to change the design of the refactored program radically. Even though the new design may be better in many ways, the developers have to invest time to understand it and this may prove to be more costly than the benefits accrued from the design improvements achieved. However, if automated refactoring is used to improve program security, then this drawback does not arise. Security only matters when the software is released and deployed and hence open to attack. Refactoring to improve security can therefore be applied as part of the final build process prior to the release and deployment of the software. The developers can continue to work with the original program whose design is familiar to them

The remainder of this paper is structured as follows. In section 2 we present an overview of the Alshammari et al. security metrics that we use in this paper. In section 3 we present the search-based refactoring platform, Code-Imp, and apply this to an industrial example in section 4. Related work is surveyed in section 5 and our conclusions and future work are presented in section 6.

2 Overview of Security Metrics

The security metrics we use in this paper are those defined by Alshammari et al. [1,3,4]. These metrics are based on the information flow within a program or software design and cover such areas as data encapsulation, cohesion, coupling, composition, extensibility, inheritance and design size. The formulae for these metrics are introduced based on the concept of classified (critical) members and classes. As defined above, a classified attribute is one whose release to an unauthorised user would constitute a breach of security. A classified method is one that interacts directly with a classified attribute. A class is said to be *critical* if it contains at least one classified attribute or method. Table 1 summarizes the Alshammari et al. security metrics that we use to guide the search-based refactoring process. In the remaining paragraphs of this section these metrics are introduced in more detail, based on descriptions by Alshammari et al. [1].

Table 1. Summary of the Alshammari et al. Security Metrics [1]

Metrics Group	Metrics Description
The cohesion-based metrics (CMAI, CAAI, CAIW, CMW)	measure the potential flow of classified attributes' values to accessor and mutator methods, designs with a large amount of classified flow.
The coupling-based metric (CCC)	measures interactions between classes and classified attributes, rewarding designs that minimise such interactions.
The composition based metric (CPCC)	penalises designs with critical classes higher in the class hierarchy, where they can be accessed by a large number of subclasses.
The extensibility-based metrics (CCE and CME)	rewards designs with fewer opportunities for extending critical classes or classified methods.
The inheritance-based metrics (CSP, CSI, CMI and CAI)	reward designs with fewer opportunities for inheriting from critical superclasses.
The design size-based metric (CDP)	rewards designs with a lower proportion of critical classes.
The data encapsulation-based metrics (CIDA, CCDA and COA)	assess the accessibility of classified attributes and methods.

Cohesion-based metrics:

- *Classified Mutator Attribute Interactions (CMAI)* is the ratio of the sum (for all classified attributes) of the number of mutator methods that may access classified attribute to the total number of possible interactions between the mutator methods and classified attributes. A mutator method is one that can set the value of an attribute.
- *Classified Accessor Attribute Interactions (CAAI)* is the ratio of the sum (for all classified attributes) of the number of accessor methods that may access classified attribute to the total number of possible interactions between the accessor methods and classified attributes. An accessor method is one that can return the value of an attribute.
- *Classified Attributes Interaction Weight (CAIW)* is the ratio of the sum (for all classified attributes) of the number of methods that may access the classified attribute to the sum (for all attributes) of the number of methods that may access the attribute.
- *Classified Methods Weight (CMW)* is the ratio of the number of classified methods to the total number of methods.

Coupling metric:

- The *Critical Classes Coupling (CCC)* metric aims to find the degree of coupling between classes and classified attributes in a given design. It is the ratio of the number of links from classes to classified attributes defined in other classes to the total number of possible links from all classes to classified attributes defined in other classes.

Composition metric:

- The *Composite-Part Critical Classes (CPCC)* metric is the ratio of the number of critical composed-part classes to the total number of critical classes.

Extensibility metrics:

- The metric *Critical Classes Extensibility (CCE)* is defined as the ratio of the number of the non-finalised critical classes to the total number of critical classes.
- The metric *Critical Methods Extensibility (CME)* is defined as the ratio of the number of the non-finalised classified methods to the total number of classified methods.

Inheritance metrics:

- *Critical Superclasses Proportion (CSP)* is the ratio of the number of critical super classes to the total number of critical classes in an inheritance hierarchy.
- *Critical Superclasses Inheritance (CSI)* is the ratio of the sum of classes that may inherit from each critical superclass to the number of possible inheritances from all critical classes in a class hierarchy.
- *Classified Methods Inheritance (CMI)* is the ratio of the number of classified methods that can be inherited in a hierarchy to the total number of classified methods in that hierarchy.
- *Classified Attributes Inheritance (CAI)* is the ratio of the number of classified attributes that can be inherited in a hierarchy to the total number of classified attributes in that hierarchy.

Design size metric:

- (CDP) Design size simply takes into account the size, i.e. the number of classes, in a given program.

Data encapsulation (Accessibility) metrics:

- *Classified Instance Data Accessibility (CIDA)* is ratio of classified instance public attributes to classified instance attributes.
- *Classified Class Data Accessibility (CCDA)* is ratio of classified class public attributes to classified class attributes.
- *Classified Operation Accessibility (COA)* is the ratio of classified public methods to classified methods.

3 Overview of Code-Imp & Refactorings

Code-Imp (Combinatorial Optimisation for Design Improvement) is a fully automated refactoring framework developed in order to facilitate experimentation in automatically improving the design of existing programs [11, 16, 19, 20]. It takes

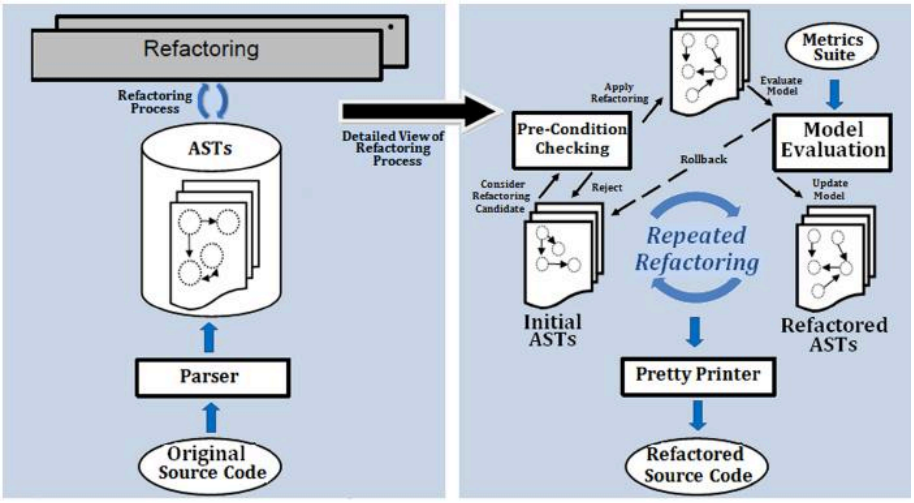

Fig. 1. Architecture of the Code-Imp automated refactoring framework (from [11])

Java version 6 source code as input and produces as output a refactored version of the program. Its output also comprises applied refactorings and metrics information gathered during the refactoring process.

Figure 1 depicts the architecture of Code-Imp. The right side of the figure shows the process of refactoring in detail. Code-Imp first extracts the initial ASTs (Abstract Syntax Trees) from the source code. Code-Imp then searches the ASTs for candidate refactorings. A refactoring is acceptable its pre- and post-conditions are satisfied and it complies with the demands of the search technique in use (in the case of a hill-climb, this means improving the quality of the design based on the metrics suite; in the case of, e.g. simulated annealing, a drop in quality may also be accepted). This process is repeated many times. After the final refactoring is applied, the ASTs are pretty printed to source code files. During the refactoring process, a rollback mechanism is supported by logging each change to the ASTs. The change history service makes it possible to perform a rollback at different levels of granularity. For example, at the finest level of granularity, individual refactorings can be reversed. At a coarser level of granularity, a composite refactoring such as a Pull Up Method (which also contains a Pull Up Field refactoring) can be reversed.

There are three aspects to the search-based refactoring process that takes place:

- the set of refactorings that can be applied;
- the type of search technique employed;
- the fitness function that directs the search.

Code-Imp currently supports 14 design-level refactorings categorized into three groups according to their scope as shown in Table 2. These are roughly

Table 2. A list of implemented refactorings in Code-Imp (from [16])

No.	Class-Level Refactorings	Description
1	Extract Hierarchy	Adds a new subclass to a non-leaf class C in an inheritance hierarchy.
2	Collapse Hierarchy	Removes a non-leaf class from an inheritance hierarchy.
3	Make Superclass Concrete	Removes the explicit *abstract* declaration of an abstract class without abstract methods.
4	Make Superclass Abstract	Declares a constructorless class explicitly abstract.
5	Replace Inheritance with Delegation	Replaces a direct inheritance relationship with a delegation relationship.
6	Replace Delegation with Inheritance	Replaces a delegation relationship with a direct inheritance relationship.
	Method-Level Refactorings	
7	Push Down Method	Moves a method from a class to those subclasses that require it.
8	Pull Up Method	Moves a method from some class(es) to the immediate superclass.
9	Decrease Method Accessibility	Decreases the accessibility of a method from protected to private or from public to protected.
10	Increase Method Accessibility	Increases the accessibility of a method from protected to public or from private to protected.
	Field-Level Refactorings	
11	Push Down Field	Moves a field from a class to those subclasses that require it.
12	Pull Up Field	Moves a field from some class(es) to the immediate superclass.
13	Decrease Field Accessibility	Decreases the accessibility of a field from protected to private or from public to protected.
14	Increase Field Accessibility	Increases the accessibility of a field from protected to public or from private to protected.

based on refactorings from Fowler's catalogue [9], though they differ somewhat in the details. Fowler's refactorings were designed specifically with the goal of design improvement in mind, whereas the goal of Code-Imp is to *explore* the design space.

The refactoring process is driven by a search technique; in this paper hill-climbing and simulated annealing are used. The simplest search technique is steepest-ascent hill-climbing, where the next refactoring to be applied is the one that produces the best improvement in the fitness function. In first-ascent hill-

climbing, the first refactoring found to improve the fitness function is applied. Hill climbing suffers from not being able to escape from local optima. Simulated annealing avoids this drawback by accepting both positive and negative moves to escape local optima with the possibility of accepting negative moves decreasing over time.

The fitness function is a measure of how "good" the program is, so the fitness function used depends on the quality that we are trying to improve. When Code-Imp is used to improve software design, the fitness function is a combination of software quality metrics. In the work described in this paper, the fitness function is based on the security metrics described in section 2. These can be combined using either a weighted-sum approach or Pareto optimality [10].

4 Case Study

In order to test the ability of automated refactoring to improve program security, we chose a sample industrial application that has a strong security dimension to refactor using Code-Imp. The application we use is Wife 6.1, one of the most commonly used open source SWIFT[3] messaging applications built in Java. It has been used by variety of financial institutions and banks [12] and comprises 3,500 lines of code and eighty classes. This is a small application, but serves as a realistic test case for assessing if automated security improvement is possible. The Code-Imp search process is guided by a fitness function based on the security metrics by Alshammari et al. [1].

Our approach is as follows. In section 4.1 we describe how the metrics to be used in the study were determined. These metrics are then combined into a fitness function using Pareto optimality. The results of refactoring Wife using this Pareto-optimal fitness function and a variety of search techniques are presented in section 4.2. In section 4.3 we analyse more closely the effect of the refactoring process on the metrics and draw some conclusions about the metrics as well as the value of refactoring to improve security. The overall conclusions to be drawn from the case study are presented in section 4.4.

4.1 Initial Metric Assessment

The first step is to annotate the Wife source code to specify the classified attributes that will be used subsequently to derive the classified methods and classes as described above. In total, five fields were marked as classified including the Bank Identifier Code (BIC) of the message and the International Bank Account Number (IBAN) of the bank account. This annotation is shown in program 1.

We then use Code-Imp to refactor this program 16 times, on each occasion using just one of the metrics from table 1 to guide the refactoring process. Our

[3] SWIFT (Society for Worldwide Interbank Financial Telecommunication) is an industry-owned cooperative providing messaging services to financial institutions.

Program 1 The Classified Field Annotation

```
@Security(classified = true)
private String bic;

@Security(classified = true)
private String iban;
```

goal is to find out which security metrics provide the best possibility to improve the security of the program. Steepest Ascent Hill-Climbing was used as the search technique, as it is simple and deterministic.

The results of this are presented in table 3. 12 of the metrics prove to be completely inert under refactoring. This is largely to do with the exact set of refactoring types that Code-Imp supports. If other refactoring types were supported, we could expect to see other metrics being affected. It is also partly dependent on the nature of the program to be refactored. Real applications, like Wife, contain certain patterns that are candidates to be targeted for refactoring, and those refactorings are likely to cause changes to the same set of security metrics and not to others. For example, if a program is not widely dependent on composition, then it is unlikely to see changes to composition metrics; the same applies to inheritance, and so on.

Table 3. Results for Running Individual Metrics against Wife Application

	Initial Value	**Final Value**	**Percentage Change**
CPCC	1.0	1.0	0.0%
CAIW	0.0037	0.0022	40.8%
CMW	0.0611	0.0611	0.0%
CSP	0.0	0.0	0.0%
CAI	0.0	0.0	0.0%
CCDA	0.0	0.0	0.0%
CMAI	0.0149	0.0149	0.0%
CCC	0.0421	0.0372	13.9%
CCE	1.0	1.0	0.0%
CIDA	0.0	0.0	0.0%
CME	1.0	1.0	0.0%
CMI	0.0	0.0	0.0%
CDP	0.1026	0.0909	13.6%
COA	0.8966	0.5172	42.3%
CAAI	0.0263	0.0263	0.0%
CMI	0.0	0.0	0.0%

The results of this experiment are clear: four metrics are improved by this process namely CAIW, CCC, CDP and COA, so we combine these in the next section to form a single fitness function.

4.2 Pareto Optimal Search

The problems associated with using weighted summation to combine ordinal-scale metrics are well documented [10]. To avoid these, we use a Pareto optimal approach to combining the four security metrics identified as promising in the previous section. In the Pareto-optimal approach, a refactoring is regarded as an improvement only if it improves at least one of the metrics and does not cause any of the others to deteriorate.

Code-Imp was run using all three search algorithms described in section 3. Steepest-Ascent Hill Climbing (HCS) was run once as it is deterministic, while First-Ascent Hill Climbing (HCF) and Simulated Annealing (SA) were each run 14 times. The small number of runs was due to the long execution time of the simulated annealing process. As this is a single-instance case study rather than a randomised experiment, the inability to use statistical inference to produce generalisable results is not critical.

The results are summarised in table 4. In the case of HCF and SA, the results presented are those from the best run.

The first observation is that HCS produces identical metric improvements here as it did when the individual metrics were used on their own. This indicates that in this example there was no conflict between the metrics, so in no case did Pareto optimality prevent a refactoring from being applied. This suggests that this set of metrics form a good combination.

HCF produces identical results to HCS, except for a minor improvement in CAIW. Simulated Annealing also produces a similar result to HCF and HCS, with the exception that the CAIW metric is improved dramatically. This is an example of simulated annealing escaping from a local optimum, and illustrates that better solutions many be found using such stochastic approaches. There was a heavy price to pay for this improvement in CAIW: while the refactoring sequences generated by the hill climbs were modest, (HCF 42 refactorings, HCS 57 refactorings), the best of the simulated annealing runs was 2194 refactorings in length. In both cases the improvement in CAIW is attributable to a weakness in this metric, as explained in Section 4.3.

4.3 Qualitative Metrics Analysis

In this subsection we delve into the metric changes in greater detail. Our aim is to understand what refactorings caused the metrics to improve and to determine if this is a real improvement or simply an artifact of the search-based refactoring process. We focus on the results of the steepest-ascent hill climb using Pareto optimality, as shown in column three of table 4.

Table 4. Security Metrics Enhancements for all Search Algorithms

	First Ascent Hill Climbing (HCF)	Steepest Ascent Hill Climbing (HCS)	Simulated Annealing (SA)
COA	42.3%	42.3%	42.3%
CAIW	41.2%	40.2%	72.6%
CCC	13.9%	13.9%	13.9%
CDP	13.6%	13.6%	13.6%

It is worth mentioning that these metrics are *ordinal*, i.e. using a scale of measurement in which data are listed in rank order but without fixed differences among the entries. This means that expressing the difference in terms of percentage may not be entirely meaningful. Nevertheless, we are assuming that the larger the difference in value, the more likely it is that a considerable improvement has taken place.

In the following paragraphs, we consider in turn each of the metrics that showed an improvement during the refactoring process.

Classified Operation Accessibility (COA): This is the ratio of classified public methods to classified methods, from the group of Data Encapsulation Metrics. The most significant change in security metrics is associated with this metric, as it undergoes a 42% enhancement from a value of 0.8966 to value of 0.5172. The change is mainly driven by the refactoring `Decrease Method Accessibility`. When applied to a public classified method, it obviously reduces the numerator and increases the denominator of the ratio, hence reducing (improving) the value of the metric. The improvement in this metric corresponds to a real improvement in program security, as it reduces the accessibility of security-critical fields.

Classified Attributes Interaction Weight (CAIW): This is the ratio of the sum (for all classified attributes) of the number of methods that may access the classified attribute to the sum (for all attributes) of the number of methods that may access the attribute, from the group of Cohesion-Based Metrics. This metric undergoes an improvement of 40% from an initial value of 0.0037 to a final value of 0.0022. Looking at the refactorings that cause the metric to improve it can be found that the refactoring `Increase Field Accessibility` is the largest contributor. This is surprising because increasing the accessibility of a field (e.g. from `protected` to `public`) would be expected to negatively impact security, if it impacts it at all. However what is happening is that reducing the accessibility of a non-classified attribute will enhance the ratio of the summation of the access to classified attributes compared to non-classified ones. This demonstrates a weakness in the CAIW metric as it detects an improvement in security although absolutely no extra protection has been achieved for the security-critical data. The other main contributor to this metric enhancement is the `Pull Up Method` refactoring which reduces the access to classified attributes in the subclass and limits them to those of the superclass. In other words moving a method to a

superclass will limit its access to classified methods in subclass. This aspect of improvement in the CAIW metric represents a real improvement in security.

Classes Design Proportion (CDP): The is the ratio of the number of critical classes to the total number of classes, from the group of Design Size Metrics. This metric is enhanced by 13.6% from an initial value of 0.1026 to an improved value of 0.0909. All improvements were caused by the `Extract Hierarchy` refactoring and were mainly due to the fact that dividing a non-classified class into two classes will generate two non-classified classes and hence enhance the ratio of classified to non-classified classes. As with CAIW, this "improvement" is meaningless and demonstrates a weakness in the metric as absolutely no extra protection to classified data is achieved as a result of the metric improvement. On the other hand, in the cases where the `Extract Hierarchy` refactoring divided a classified class into one classified and one non-classified class, the improvement in CDP was real. Breaking a critical class into two classes with only one having access to classified data means that data is rendered accessible to fewer methods and is therefore more secure.

Critical Classes Coupling (CCC): This is the ratio of the number of links from classes to classified attributes defined in other classes to the total number of possible links from all classes to classified attributes defined in other classes, from the group of Coupling-Based Metrics. The behaviour of CCC is very similar to that of CDP both in terms of the percentage of enhancement (13.9%) and the effect of the `Extract Hierarchy` refactoring on its value. It is interesting to note note that every time CDP was affected by an `Extract Hierarchy` refactoring, CCC was affected by a proportional value. Both the metric weakness and the recommendation for the CDP metric above are applicable.

4.4 Case Study Conclusions

The key research question we wish to investigate is whether automated refactoring can be used to improve program security. 12 out of the 16 Alshammari et al. metrics we tested proved to be inert under the refactorings we applied, but this is a consequence of the refactoring suite we use. For the four metrics that were affected by the refactoring, in each case some of the metric improvement corresponded to real improvement in program security, as detailed in the *Real* column of Table 5. This is a positive result in terms of the ability of the search-based refactoring approach to improve security.

As a by-product of our analysis, we made several other interesting discoveries about the metrics we used to guide the refactoring process. These were derived from where the improvement in the security metrics was found not to represent a true improvement in security, as detailed in the *Artificial* column of Table 5.

CAIW is a poorly formed metric that rewards the increasing of the accessibility of a non-classified attribute, even though this obviously has no impact on program security, and in fact is an example of poor object-oriented design. CDP was also found to be poorly-formed as it rewards the splitting of non-classified classes which again has no impact on program security, and is again is an example of poor object-oriented design. The ability of search-based refactoring to

Table 5. Security Metrics Enhancements for Steepest-Ascent Hill Climbing

	Improvement	**Real**	**Artificial**
COA	42.3%	42.3%	0.0%
CAIW	40.2%	14.5%	25.7%
CCC	13.9%	3.5%	10.4%
CDP	13.6%	2.3%	11.3%
Average	27.5%	15.6%	11.9%

pinpoint metric weaknesses like this was also demonstrated in previous work that used the QMOOD metrics suite [5] to guide search-based refactoring [20].

5 Related Work

This work merges two research strands: software security metrics and search-based refactoring.

The traditional approach to measuring software security is to measure the number of security-related bugs found in the system or the number of the system is reported in security bulletins [8]. More recent work has focussed on measuring properties of the software design that are related to security [1, 8].

As well as proposing the metrics suite using in this paper [1, 4], Alshammari et al. developed a hierarchical approach to assessing security [2]. They chose two principles to measure the security of designs from the perspective of information flow: *grant least privilege* [6] which means that each program component will have access to *only* the parts that it legitimately requires, and *reduce attack surface* [7] which means to reduce the amount of code running with an unauthenticated user. In a follow-up study that is more closely related to our work [3], Alshammari et al. looked at the effect on their security metrics of applying refactorings to program designs. While this can provide static insight into the properties of the refactorings, it does not automate the improvement of the program as our approach does.

Smith and Thober [22] try to expose information flow security by using code refactoring to partition a system into high security and low security components, where high security components can take high or low security input but cannot send output to low security components unless this output is investigated and approved (i.e. declassified). The program needs to be analysed first to identify high and low security components, and then refactored to isolate the high security components. Metrics are not used to measure code security, instead a manual (or partly-automated) partitioning is used to isolate code with various security levels. Full automation is impossible, as both partitioning and declassification need to be performed by developers knowledgeable in the code. By contrast, our approach requires minimal developer input.

The main application of search-based refactoring has been to automate the improvement of a program's design. O'Keeffe and Ó Cinnéide [20] propose an

automated design improvement approach to improve software flexibility, understandability, and reusability based on the QMOOD quality model [5]. They report a significant improvement in understandability and minimal improvement in flexibility, however the the QMOOD reusability function was found to be unsuitable for automated design improvement. Other work by the same authors [18] investigates if a program can be automatically refactored to make its metrics profile more similar to that of another program. This approach can be used to improve design quality when a sample program has some desirable features, such as ease of maintenance, and it is desired to achieve these features in another program.

Seng et al. [21] propose an approach for improving design quality by moving methods between classes in a program. They use a genetic algorithm with a fitness function based on coupling, cohesion, complexity and stability to produce a desirable sequence of move method refactorings. Improving design quality by moving methods among classes was also investigated by Harman and Tratt [10]; their key contribution is the use a Pareto optimal approach to make combination of metrics easier. Jensen and Cheng [13] use refactoring in a genetic programming environment to improve the quality of the design in terms of the QMOOD quality model. Their approach was found to be capable of introducing design pattern through the refactoring process, which helps to change the design radically. Kilic et al. explore the use of a variety of population-based approaches to search-based parallel refactoring, finding that local beam search could find the best solutions [14]. Search-based refactoring has also been used to improve other aspects of software, e.g. to improve program testability [17]. The work presented in this paper is the first attempt to use search-based refactoring to improve software security.

6 Conclusions and Future Work

This paper builds on previous work that shows that refactoring can have a substantial effect on security metrics when applied to a software design [3]. We extend this work by using the search-based refactoring platform, Code-Imp, guided by security metrics to test if the security of source code can be improved in an automated fashion. In our study of an industrial software application, Wife, we achieved an overall real improvement of 15.5% in the metrics affected by the refactoring. This improvement is obtained by a fully-automated search-based refactoring process that requires no developer input other than the annotation of the classified fields. This security gain at such little cost indicates the value and potential of the approach.

Previously search-based refactoring has been used mainly to refactor a program in order to improve its design [10, 11, 19]. This creates the difficult problem of explaining the new design to the developer. In using search-based refactoring to improve security, such problems do not arise. The process is applied to a program/library just prior to it being released; the developers continue to work with the original version of the code.

As summarised in section 4.4, we also demonstrated that some of the Alshammari et al. security metrics are poorly formed and require reworking in order to more truly reflect program security. These are issues to be addressed by security researchers.

Future work involves extending Code-Imp with new refactorings that can impact the inert security metrics, e.g. `Extract Method`, as well as adding refactorings that have a specific security dimension, e.g. `Make Class Final`. One weakness of the security metrics employed in this paper is that they do not define clearly the type of attack they protect the program from. If security metrics were to specify this attack profile, then security test cases could be created that exploit the security vulnerabilities injected into a sample program. Then it could be tested if our refactoring approach could actually fix the security vulnerabilities. This would a be a more robust demonstration of construct validity than the examination of the changes brought about by refactoring that we performed in this paper.

While our experiences in refactoring Wife have demonstrated the potential value of search-based refactoring to improve program security, further experiments with larger applications, extra refactorings and more refined metrics are required to fully explore the potential of this approach.

Acknowledgment

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855. The authors would like to thank Iman Hemati Moghadam for his help in performing the experiments with the Code-Imp refactoring platform. Shadi Ghaith gratefully acknowledges the support of IBM in this work.

References

1. B. Alshammari, C. Fidge, and D. Corney. Security metrics for object-oriented class designs. In *Proceedings of the International Conference on Quality Software*, pages 11–20. IEEE, 2009.
2. B. Alshammari, C. Fidge, and D. Corney. A hierarchical security assessment model for object-oriented programs. In *Proceedings of the International Conference on Quality Software*, pages 218–227, Los Alamitos, USA, 2011. IEEE.
3. B. Alshammari, C. J. Fidge, and D. Corney. Assessing the impact of refactoring on security-critical object-oriented designs. In J. Han and T. D. Thu, editors, *Proceedings of the Asia Pacific Software Engineering Conference*, pages 186–195. IEEE Computer Society, 2010.
4. B. Alshammari, C. J. Fidge, and D. Corney. Security metrics for object-oriented designs. In J. Nobel and C. J. Fidge, editors, *The 21st Australian Software Engineering Conference*, pages 55–64, Hyatt Regency, Auckland, June 2010. IEEE.
5. J. Bansiya and C. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28:4–17, Jan 2002.
6. M. A. Bishop. *The Art and Science of Computer Security.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

7. C. Blackwell. A security architecture to protect against the insider threat from damage, fraud and theft. In *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW '09, pages 45:1–45:4, New York, NY, USA, 2009. ACM.
8. I. Chowdhury, B. Chan, and M. Zulkernine. Security metrics for source code structures. In *Proceedings of the fourth international workshop on Software engineering for secure systems*, SESS '08, pages 57–64, New York, NY, USA, 2008. ACM.
9. M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999.
10. 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*, GECCO '07, pages 1106–1113, New York, NY, USA, 2007. ACM.
11. I. Hemati Moghadam and M. Ó Cinnéide. Code-Imp: a tool for automated search-based refactoring. In *Proceedings of the 4th workshop on Refactoring tools*, WRT '11, pages 41–44, New York, NY, USA, 2011. ACM.
12. http://www.prowidesoftware.com. Wife swift application. In *Wife Swift Application.* Prowide Open Source SWIFT, 2012.
13. A. Jensen and B. Cheng. On the use of genetic programming for automated refactoring and the introduction of design patterns. In *Proceedings of the Conference on Genetic and Evolutionary Computation*, page 1341–1348. ACM, July 2010.
14. H. Kilic, E. Koc, and I. Cereci. Search-based parallel refactoring using population-based direct approaches. In *Proceedings of the 3rd international Conference on Search Based Software Engineering*, SSBSE'11, pages 271–272, 2011.
15. G. McGraw. *Software Security: Building Security In.* Addison-Wesley Professional, 2006.
16. I. H. Moghadam and M. Ó Cinnéide. Automated refactoring using design differencing. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 43–52. IEEE Computer Society, 2012.
17. M. Ó Cinnéide, D. Boyle, and I. Hemati Moghadam. Automated refactoring for testability. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops*, March 2011.
18. M. O'Keeffe and M. Ó Cinnéide. Automated design improvement by example. In *Proceeding of the Conference on New Trends in Software Methodologies, Tools and Techniques*, pages 315–329, 2007.
19. M. O'Keeffe and M. Ó Cinnéide. Search-based refactoring: an empirical study. *Journal of Software Maintenance and Evolution*, 20(5):345–364, 2008.
20. M. O'Keeffe and M. Ó Cinnéide. Search-based refactoring for software maintenance. *Journal of Systems and Software*, 81(4):502–516, 2008.
21. O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *GECCO'2012*, pages 1909–1916, Seattle, Washington, USA, July 2006. ACM.
22. S. F. Smith and M. Thober. Refactoring programs to secure information flows. In *Proceedings of the Workshop on Programming Languages and Analysis for Security*, PLAS '06, pages 75–84, New York, NY, USA, 2006. ACM.