
© Copyright 2009 The Institute of Electrical and Electronics Engineers
Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Security Metrics for Object-Oriented Class Designs

Bandar Alshammari¹, Colin Fidge² and Diane Corney³

School of Information Technology
Queensland University of Technology, Brisbane, Australia
¹b.alshammari@student.qut.edu.au
²c.fidge@qut.edu.au
³d.corney@qut.edu.au

Abstract

Measuring quality attributes of object-oriented designs (e.g. maintainability and performance) has been covered by a number of studies. However, these studies have not considered security as much as other quality attributes. Also, most security studies focus at the level of individual program statements. This approach makes it hard and expensive to discover and fix vulnerabilities caused by design errors. In this work, we focus on the security design of an object-oriented application and define a number of security metrics. These metrics allow designers to discover and fix security vulnerabilities at an early stage, and help compare the security of various alternative designs. In particular, we propose seven security metrics to measure Data Encapsulation (accessibility) and Cohesion (interactions) of a given object-oriented class from the point of view of potential information flow.

Keywords:
Quality; Security; Metrics; Design Principles; Refactoring

1. Introduction

System requirements have been classified into two categories: functional and non-functional requirements (NFRs) [1]. Security is considered part of a system’s non-functional requirements. Such software quality attributes include, in addition to security, maintainability, performance, reusability, and reliability [1]. Most of these attributes have been studied and measured extensively. Security measurements have been defined to assess security at the system level [2] and the level of implementation code [3]. However, measuring security at the design phase, based on typical design artifacts, has received little attention.

This paper proposes a new set of metrics which are capable of assessing the security quality of object-oriented classes. Previous studies have agreed that taking security into account from the early stages of a system’s development should have a significant impact on decreasing many software vulnerabilities [1] [4]. Therefore, we have defined a set of seven security metrics that can help programmers assess class designs from an information flow perspective.

The approach taken in this work was to define these metrics based on the quality properties specified by Bansiya et al. [5]. We have chosen those properties which are related to individual object-oriented classes: data encapsulation and cohesion. Our proposed metrics aim to measure any potential information flow which could occur through objects instantiated from these classes. However, in order to measure the impact of these properties on security, we need annotated class diagrams. In our case, we use UMLsec and SPARK’s annotations. UMLsec’s annotations identify confidential data [6] while SPARK’s annotations express the information flow relations between methods and attributes of a given class [7], which are normally not shown in class designs. Once the metrics’ results are identified for a number of alternative designs, it is easy to choose the most secure design. This can be done by either comparing the overall results or by choosing the results which satisfy a certain security design principle.

2. Related Work

Most current studies on software security admit that there is no such thing as a completely secure program, but there are nevertheless various ways of reducing security risks and vulnerabilities [8] [9]. One of these is the enforcement of security in the implementation.
Several projects have been conducted to investigate information flow through computer program code. This has been studied using several approaches, including type analysis [10] and data/control-flow analysis [11].

Another approach is to enforce security at early phases of the software development lifecycle such as at the design phase. One of the earliest studies in this area was the development of software security design principles by Saltzer and Schroeder [12]. These principles were intended as guidance to help develop secure systems, mainly operating systems. Bishop’s [13] and McGraw’s [14] texts identified several similar security design principles. However, these principles were not capable of quantifying the security levels of programs. Thus, there is a need for security metrics based on these principles to objectively measure the security of a given program directly from its design artifacts.

Defining software security metrics is another way of reducing program security risks and vulnerabilities. An existing approach which is used by programmers to assess the level of security of given program code is based on the identification of vulnerabilities [15] [9]. A study conducted by Chowdhury et al. [3] defined a number of security metrics that assess the security of a given program based on code inspections. These metrics need full implementations of the system to assess its security. This approach makes it hard to fix security errors at the design time and is expensive in terms of time and resources.

In addition, measuring the security of the system’s architecture has been done by Manadhata et al. [2]. This study focused on the system’s ‘attack surface’. Similarly, a study that defined design metrics which measure certain software quality attributes was conducted by Bansiya [16]. He identified an approach to improve the Quality Model for Object-Oriented Design (QMOOD) [5]. The model aims to measure the quality of various object-oriented design attributes such as reusability, flexibility, and functionality based on their relevance to certain quality design properties (e.g. abstraction, cohesion, and coupling). Even though the study covered most design quality attributes, it did not consider security. Since security is a quality requirement [1], developing a number of security metrics based on the QMOOD quality design properties is the best option for designing metrics.

Defining a set of metrics which evaluates the security of a given program based on its design artifacts rather than its source code would reduce the cost of fixing security design vulnerabilities by detecting these vulnerabilities at an early stage. In this paper we define seven new security design metrics. They can be used to compare different designs for the same program and identify the best design for a certain security design principle. They do this by identifying potential information flow based on analysing the software quality properties defined in the QMOOD.

3. Assumptions and Annotations

Our security design metrics are designed to be capable of quantifying the security level of a given object-oriented class. They are different from typical “code complexity” metrics, which measure syntactic properties of the code such as the number of variables and lines of code. Instead, our metrics measure potential information flow properties within a given class based on its design. This measurement is a comparative one. It can be used to compare various alternative designs of the same class with respect to their security properties.

The metrics have been scaled to all fit with the range 0 to 1. A low value is desired for each. Their results show which alternative designs will increase or decrease the security of a given class with regard to a specific software security design principle (e.g. Least Privilege, Reduce Attack Surface, etc [13] [17]).

Our metrics at this stage are concerned with the properties of individual object-oriented classes. Two properties are covered: the accessibility of, and interactions within, classes. (Other properties of multiple classes, such as inheritance, coupling, and extensibility, will be considered in future work.)

To apply our metrics to a given design, we assume that system designers will accurately provide annotated UML class diagrams using UMLsec and SPARK’s annotations. UMLsec [6] is an extension of the Unified Modeling Language which labels objects as ‘critical’ if they consist of data which can be of a security risk at any point. It also associates a ‘secrecy’ tag with data which needs to be kept confidential [6]. Our metrics consider classified data as that which is defined as “secrecy” in UMLsec.

On the other hand, SPARK is a programming language for security-critical code in which the programmer may annotate subroutines with the intended data flow between variables and parameters. The SPARK compiler then performs a data-flow analysis to confirm that the code does indeed have the characteristics the programmer intended. SPARK’s annotations consist of a “derives from” block which explains how the value of a certain variable or return value is potentially derived from the value of another method parameter or variable [7].

Some terminology associated with our metrics is shown in Table 1.
several techniques to reduce the attack surface size of a system. Howard [17] has identified some requirements of the security principle of reducing the attack surface. The principle of "least privilege" is described as "programs and users should run with the least privilege to complete their job" [12]. The main advantage of this principle is to minimize the interactions among privileged programs [12]. To adhere to this principle, systems must restrict the privileges of their users to the least possible. In a class design, this means the design whose methods can do the fewest possible actions is the most secure. In our case, a class whose methods interact with the fewest possible classified attributes would be a secure design with respect to this principle.

The reduce attack surface principle aims to limit access to secret data. Howard [17] has identified several techniques to reduce the attack surface size of a given system including reducing the amount of running code and access to entry points. For our purposes, this means a class design should have the fewest possible accessible methods each with the fewest number of parameters which can affect classified attributes needed for necessary tasks. A class which has less accessibility to classified methods that interact with as few classified attributes as possible would satisfy the requirements of the security principle of reducing the attack surface.

4. Security Design Metrics

This section consists of three parts. Part one explains the security design principles relevant to our metrics and our analysis of these principles. The other two parts explain our security design metrics.

4.1 Relevant Security Design Principles

This section describes the security design principles covered by our security design metrics. As mentioned previously, a number of studies have presented several design principles for developing secure systems [12] [13] [14]. In our study, we have chosen two principles to measure the security of designs from the perspective of information flow: least privilege [13] and reduce attack surface [17].

The principle of least privilege is described as "programs and users should run with the least privilege to complete their job" [12]. The main advantage of this principle is to minimize the interactions among privileged programs [12]. To adhere to this principle, systems must restrict the privileges of their users to the least possible. In a class design, this means the design whose methods can do the fewest possible actions is the most secure. In our case, a class whose methods interact with the fewest possible classified attributes would be a secure design with respect to this principle.

4.2 Security Accessibility Metrics

Metrics under this category aim to measure the accessibility level of attributes and methods in a particular class from an access modifier perspective. Access modifiers are associated with each class, method, and attribute to control their accessibility [18]. These modifiers include: public, protected, and private.

Maruyama et al. [19] have investigated how changes to access modifiers could change the security characteristics of a given program. Their work shows which refactoring rules could change a class’s accessibility level and therefore changes its security level. However, this approach doesn’t quantify the impact of these changes on the security level of a given program. Our accessibility metrics are similar to the one used by Bansiya [16] to measure the encapsulation property of a class, called the Data Access Metric (DAM). DAM is measured as the ratio of the number of private (protected) attributes to the total number of attributes in a declared class [16].

Bansiya [16] also included another metric to measure the accessibility of methods, called the Operation Access Metric (OAM). OAM is defined as the ratio of the number of public methods to the total number of methods in a class [16].

Our security accessibility metrics statically measure the potential flow of information from an accessibility perspective for an individual object-oriented class. These metrics only consider attributes and methods declared as classified since they are the ones which need to be kept secret. We divide these metrics for individual classes into three kinds of accessibility: instance attributes; class attributes; and methods.

**Table 1. Metrics Terminology**

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classified Attribute</td>
<td>An attribute which is defined in UMLsec as secrecy.</td>
</tr>
<tr>
<td>Instance Attribute</td>
<td>An attribute whose value is stored by each instance of a class [18].</td>
</tr>
<tr>
<td>Class Attribute</td>
<td>An attribute whose value is shared by all instances of that class [18].</td>
</tr>
<tr>
<td>Classified Methods</td>
<td>A method which interacts with at least one classified attribute.</td>
</tr>
<tr>
<td>Unclassified Method</td>
<td>A method which doesn’t interact with any classified attributes.</td>
</tr>
<tr>
<td>Mutator</td>
<td>A method that can set the value of an attribute [18].</td>
</tr>
<tr>
<td>Accessor</td>
<td>A method that can return the value of an attribute [18].</td>
</tr>
</tbody>
</table>

4.3 Security Design Metrics

This section describes the security design principles relevant to our metrics and our analysis of these principles. The other two parts explain our security design metrics.
surface’. This means a higher possibility for confidential data to be exposed to unauthorized parties. Aiming for lower values of this metric adheres to the security principle of reducing the attack surface [17].

Consider a set of classified attributes in class C as $CA = \{ca_1, ..., ca_n\}$ and its classified instance public attributes as $CIPA = \{cipa_1, ..., cipa_n\}$ such that $CIPA \subseteq CA$. Then:

$$CIDA(C) = \frac{|CIPA|}{|CA|}$$

(Classified Class Data Accessibility (CCDA))

This metric measures the direct accessibility of classified class attributes of a particular class. (In UML class diagrams, class attributes are underlined where they are declared.) This metric aims to protect the classified internal representations of a class, i.e., class attributes, from direct access. It is defined as follows: “The ratio of the number of classified class public attributes to the number of classified attributes in a class”. This metric is calculated by dividing the number of public classified class attributes of a given class by its total number of classified attributes. The result shows the ratio of classified class attributes which are directly accessible from outside its class. Higher values mean that confidential data of that class has a higher chance of being exposed to unauthorized parties. This metric contributes towards measuring the attack surface size of a given program’s classified class attributes. Thus, lower values of this metric enforce the security principle of reducing the attack surface [17].

Consider a set of classified attributes in class C as $CA = \{ca_1, ..., ca_n\}$ and the classified class public attributes as $CCPA = \{ccpa_1, ..., ccpa_n\}$ such that $CCPA \subseteq CA$. Then:

$$CCDA(C) = \frac{|CCPA|}{|CA|}$$

(Classified Operation Accessibility (COA))

This metric is the ratio of the accessibility of public classified methods of a particular class. We define it as: “The ratio of the number of classified public methods to the number of classified methods in a class”. It is calculated by dividing the number of classified methods which are declared as public in a given class by its total number of classified methods. This value also indicates the size of the attack surface of a given class. It aims to protect the internal operations of a class which interact with classified attributes from direct access. Lower values of this metric would reduce potential information flow of classified data which could be caused by calling public methods. This metric measures the potential attack surface size exposed by classified methods [17].

Consider a set of all classified methods in class C as $CM = \{cm_1, ..., cm_n\}$ and the classified public methods in that class as $CPM = \{cpm_1, ..., cpm_n\}$ such that $CPM \subseteq CM$. Then:

$$COA(C) = \frac{|CPM|}{|CM|}$$

4.3 Security Interactions Metrics

Our interactions metrics are defined to measure the impact of class interactions between methods and attributes on the security of that class. They are defined in a way similar to the cohesion metric defined by Briand et al. [20]. Their cohesion metric is defined as the ratio of the number of methods’ interactions with attributes in the program code to the maximum number of methods’ interactions with attributes [20]. However, our interaction metrics instead measure the potential flow of information caused by methods’ and attributes’ interactions in a given class. We classify methods of a class for these metrics as follows: Classified Mutators (setters/writers/constructors); Classified Accessors (getters/readers); or Unclassified methods. These metrics are divided into four parts: the interactions of mutators with classified attributes; the interactions of mutators with classified attributes which means a lower weight of classified methods.

Classified Mutator Attribute Interactions (CMAI)

This metric measures the interactions of mutators with classified attributes in a class. We define this metric as: “The ratio of the number of mutators which may interact with classified attributes to the possible maximum number of mutators which could interact with classified attributes”. To calculate this metric, we first need to find out in how many places in the design/program classified attributes could be mutated. Then, we divide this number by the total number of possible ways of mutating these classified attributes. The result is a ratio which can be used to indicate the potential interactions between mutators and classified attributes. Higher interaction means stronger cohesion between mutators and classified attributes within a given class, and consequently more privileges are given to mutators on classified attributes. Conversely, lower values indicate weaker cohesion between mutators and classified attributes which means a lower
chance of classified information flow from mutators. With regard to the security principles, a lower value allows fewer privileges over confidential data and therefore adheres to the least privilege principle [13].

Consider a set of mutator methods in class C as $MM, i \in \{1, \ldots, mm\}$ and the classified attributes as $CA, j \in \{1, \ldots, ca\}$. Let $\alpha(CA_j)$ be the number of mutator methods which may access classified attribute $CA_j$. Then, CMAI for mutator methods for class C can be expressed as follows:

$$
CMAI(C) = \frac{\sum_{j=1}^{ca} \alpha(CA_j)}{|MM| \times |CA|}
$$

Classified Accessor Attribute Interactions (CAAI)

This metric measures the interactions of accessor methods with classified attributes in a class. We define this metric as: “The ratio of the number of accessor methods which may interact with classified attributes to the possible maximum number of accessor methods which could have access to classified attributes”. This metric is calculated in a similar way to the CMAI metric by first finding out in how many parts of the design/program classified attributes could be accessed. Then, this number is divided by the total number of possible ways of accessing these classified attributes. This results in a ratio which directly shows the potential interactions between accessors and classified attributes. Higher interaction means stronger cohesion between accessors and classified attributes within a given class. Similar to mutators, weak cohesion is desirable to reduce any potential flow of classified data caused by accessors. Weak cohesion also indicates fewer privileges are given to accessors over classified attributes. This would reduce the chance of potential flow of classified data to adversaries. Moreover, lowering the value of this metric would lower privileges of accessors over classified attributes and thus satisfy the security principle of least privilege [13].

Consider a set of accessor methods in class C as $AM, l \in \{1, \ldots, am\}$ and classified attributes as $CA, j \in \{1, \ldots, ca\}$. Let $\beta(CA_j)$ be the number of accessor methods which may access attribute $CA_j$. Then, CAAI for class C can be calculated as:

$$
CAAI(C) = \frac{\sum_{j=1}^{ca} \beta(CA_j)}{|AM| \times |CA|}
$$

Classified Attributes Interaction Weight (CAIW)

This metric is defined to measure the interactions with classified attributes by all methods of a given class. We define this metric as: “The ratio of the number of all methods which may interact with classified attributes to the total number of all methods which could have access to all attributes”. This metric is calculated by finding the number of methods of a given class which may interact with classified attributes, and dividing this number by the total number of potential interactions with all attributes in that class. The importance of this metric is that it shows how many potential class interactions are dependent on classified attributes.

This is another metric which measures the privileges of class methods over classified data. However, this metric differs from the previous ones as it shows the overall privileges by a class’ methods over classified attributes. The higher the value of this metric for a given class the more privileges are given to this class’ methods over classified attributes, and therefore the less that class adheres to the security principle of least privilege [13].

Consider a set of attributes in class C as $A, i \in \{1, \ldots, a\}$ and a set of classified attributes as $CA, j \in \{1, \ldots, ca\}$. Let $\gamma(CA_j)$ be the number of methods which may access classified attribute $CA_j$. Let $\delta(A_i)$ be the number of methods which may access attribute $A_i$. Then, CAIW can be computed as:

$$
CAIW(C) = \frac{\sum_{j=1}^{ca} \gamma(CA_j)}{\sum_{i=1}^{a} \delta(A_i)}
$$

Classified Methods Weight (CMW)

This metric is defined to measure the weight of methods in a class which potentially interact with any classified attributes in a particular class. We define this metric as: “The ratio of the number of classified methods to the total number of methods in a given class”. From this definition, we can calculate this metric by initially summing the number of methods which may interact in any form with classified attributes in a class. Then, we divide this number by the total number of methods in that class. This metric can directly measure the attack surface size of a given class based on its operations over confidential data. This differs from our previous attack surface metrics as it doesn’t focus on accessibility but instead it focuses on the interaction weight of classified methods. Higher values of this metric indicate more classified operations are offered by the given class. This leads to a higher chance of information flow of classified data by calling the class’s methods and violations of the security principle of reducing the attack surface [17].

Consider a set of methods in a class C as $M = \{m_1, \ldots, m_n\}$ and the classified methods as $CM = \{C_{m_1}, \ldots, C_{m_n}\}$.
expressed as: 
\[
CMW(C) = \frac{|CM|}{|M|}
\]

5. Metrics Case Study

The following case study illustrates how our software security design metrics are used. They can be applied once a UML class diagram of a single class is constructed. This class diagram must include UMLsec and SPARK’s annotations in addition to the standard elements of a UML class diagram.

The case study consists of class diagrams of an original design and several refactored versions of the class and the metrics results. There are a number of assumptions associated with these metrics:

- Any method that changes the value of an attribute is a mutator.
- Any method that returns the value of an attribute is an accessor.
- A method can be both a mutator and an accessor.
- Constructors are considered to be a special type of mutator if they can set any attributes.
- Some object oriented languages allow methods to have parameters which return values (such as “out parameters” in C#); we consider these methods as accessors.

5.1 UML Class Diagrams

The class diagrams in this section show various designs of a single class of a given system. The ContactNos class is responsible for storing information about a person’s contact numbers. Its attributes consist of a person’s name, phone area code, and phone extension number which is also their office number. Its operations are responsible for mutating and accessing these details once they have been requested. Details of a person’s office number, and hence their area code, are meant to be kept secret.

To illustrate the capabilities of our metrics, we apply them to seven refactored versions of the ContactNos design. We assume that Figure 1 shows the original ContactNos UML class diagram. Figures 2 to 7 show refactored versions of the original design using one or more the refactoring methods defined by Fowler [21]. (We are aware that the original design can be refactored into a design which consists of more than one class. As mentioned previously, we are only considering refactoring designs for single classes in this paper).

For instance, Figure 2 shows a design which has been constructed after applying a number of refactoring steps to the original design. It can be seen that ContactNos’s attributes are declared private unlike Figure 1 in which they were public. This is done by using the Encapsulate Field refactoring rule. Another change is the introduction of a mutator which mutates a person’s contact details and an accessor which returns a person’s contact details. This has been done by using a refactoring rule called Inline Method which combines the outputs of more than one method into one. Figure 2 keeps the rest of the methods in Figure 1 unchanged except for declaring them as private.

Another design is Figure 3 which has the same changes as in Figure 2. In addition, it has changed the parameter type of the class’s mutators to integers. This change, consequently, has led to the introduction of a new method ParseIntToString which is neither a mutator nor an accessor. This method’s job is to parse integers to strings in order to be compatible with the class’s attributes type.

Figure 4 shows a design which is similar to Figure 3 except it has used the refactoring rule Extract Method. Extract Method has been applied to the mutators and accessors of areaCode and officeNo attributes. This was done to mutate and access their values separately while making their methods private.

Figure 5 has declared the class’s attributes as private by using the refactoring rule Encapsulate Field. It has also used the refactoring rule Inline Method when compared to the original class design. This step has led to introducing just one mutator and one accessor to set and get the values of all attributes.

Figure 6 has the same changes as in Figure 5 but has also combined both of the telephone number attributes to one attribute called teleNo which is declared as “secrecy”. This refactoring rule is not shown in the literature but we call it “Inline Field”.

Figure 7 kept the previous changes to the attributes but instead it has provided the class with a mutator and accessor for each of the two attributes. It can be seen from the previous design that the class has kept the same capabilities as in the original design. However, what differentiates them is how the classes are internally structured. Thus, our security design metrics are designed to be capable of showing these changes.

5.2 Metrics Results and Representations

Table 2 shows the results of applying our metrics to the seven designs shown below. To make it easier to understand and compare these results, we also show them in radar charts (Figures 8 to 11). This allows us to easily compare which aspects of the system are most secure and which ones are not. Given that lower values
Figure 1. ContactNos 1 class diagram

Figure 2. ContactNos 2 class diagram

Figure 3. ContactNos 3 class diagram

Figure 4. ContactNos 4 class diagram

Figure 5. ContactNos 5 class diagram

Figure 6. ContactNos 6 class diagram

Figure 7. ContactNos 7 class diagram
Table 2. Security Design Metrics Results

<table>
<thead>
<tr>
<th>Design</th>
<th>CIDA</th>
<th>CCDA</th>
<th>COA</th>
<th>CMAI</th>
<th>CAAI</th>
<th>CAIW</th>
<th>CMW</th>
</tr>
</thead>
<tbody>
<tr>
<td>ContactNos 1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.5</td>
<td>0.5</td>
<td>0.8</td>
<td>0.75</td>
</tr>
<tr>
<td>ContactNos 2</td>
<td>0</td>
<td>0</td>
<td>0.33</td>
<td>0.5</td>
<td>0.5</td>
<td>0.67</td>
<td>0.75</td>
</tr>
<tr>
<td>ContactNos 3</td>
<td>0</td>
<td>0</td>
<td>0.33</td>
<td>0.625</td>
<td>0.625</td>
<td>0.67</td>
<td>0.67</td>
</tr>
<tr>
<td>ContactNos 4</td>
<td>0</td>
<td>0</td>
<td>0.33</td>
<td>0.5</td>
<td>0.5</td>
<td>0.8</td>
<td>0.67</td>
</tr>
<tr>
<td>ContactNos 5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.67</td>
<td>1</td>
</tr>
<tr>
<td>ContactNos 6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>ContactNos 7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Figure 8. ContactNos 1 metrics results
Figure 9. ContactNos 2 versus ContactNos 3
Figure 10. ContactNos 4 versus ContactNos 5
Figure 11. ContactNos 6 versus ContactNos 7

of each metric are considered more secure, designs whose charts are closer to the centre are considered best. It can be seen that ContactNos 1 from Figure 8 has the most insecure design with regard to the accessibility of classified instance attributes (CIDA) while the other designs are equivalent in that measure.

This is due to declaring the classified attributes in the ContactNos 1 class diagram as public. However, all designs have the same classified class attributes accessibility (CCDA) measure since they don’t have any classified class attributes.
In terms of classified methods accessibility, the class diagrams of ContactNos 1, 5, 6, and 7 declare all of their classified methods as public which resulted in these designs being the most insecure designs for the Classified Operation Accessibility (COA) metric. The ContactNos 2, 3, 4 class diagrams declare only a third of their classified methods as public which resulted in these designs being the most secure ones in this case.

On the other hand, ContactNos 5 and 6 show the most insecure designs with regard to the cohesion between mutators (CMAI) and accessors (CAAI) with classified attributes. This is because the ContactNos 5 and 6 class diagrams allow their mutators and accessors to interact with all of their classified attributes. Conversely, ContactNos 1, 2, 4, and 7 allow their mutators and accessors to interact with the fewest classified attributes, thus these designs are the most secure in this regard.

With regard to the weight of interactions with classified attributes (CAIW), the ContactNos 1 and 4 class diagrams have the highest value of methods’ interactions with classified attributes which causes these designs to be the most insecure for this metric. By contrast, ContactNos 6 and 7 have the least value of methods’ interactions with classified attributes which makes these designs to be the most secure in this respect.

The dependence of a class on classified methods (CMW) is shown to be most secured in the design of ContactNos 7 since it has the least number of classified methods. ContactNos 5 and 6 are the most insecure designs for this metric because they have the highest number of classified methods. In fact, all methods in ContactNos 5 and 6 are classified which make these designs totally dependent on classified methods.

5.3 Metrics Analysis

The simplest way of comparing these designs is to look at the radar charts of their designs to decide which design is the most secure. The design with the lowest values for all metrics is the most secure. This will lead us to decide that ContactNos 7’s design is the most secure for all aspects except for metric COA.

However, different systems may have different security requirements. Thus, we can interpret the results of our metrics with regard to different security design principles. These characteristics, in our case, will be related to the accessibility and interactions of a given design. An example of this approach would be to consider the previously mentioned security principles in Section 4.1: least privilege and reduce the attack surface. Based on the requirements of the least privilege principle, the design of ContactNos 7 would be the most secure since the metrics of CMAI, CAAI, and CAIW have yielded the lowest value. On the other hand, from the requirement of reducing the attack surface the design of ContactNos 3 would be the most secure because the CIDA, CCDA, COA, and CMW metrics have yielded the lowest values.

In general, we conclude that the metrics on the top of the charts: CMW, CIDA, CCDA, and COA are the ones which mostly contribute to the principle of reducing the attack surface. On the other hand, metrics at the bottom of the charts: CMAI, CAAI, and CAIW are the ones mostly associated with the principle of least privilege.

We can also consider the security metrics from the point of view of which refactoring methods are the most appropriate to accomplish the requirements of a certain security design principle. It can be seen from these designs that using the Encapsulate Field and Extract Method and Extract Field refactoring rules made the designs secure from the perspective of the least privilege principle. Conversely, using refactoring techniques such as Encapsulate Field and Inline Method and Inline Field can satisfy the requirements of the principle of reducing the attack surface.

Of course the most secure design is one which has a lower value with regard to all of these security metrics. Unfortunately, we usually face a trade off because reducing one metric often results in increasing another.

6. Discussion

To develop these metrics, we started by studying the rules for writing secure code for a given program [9] [22]. We then had to decide how to choose which rules are relevant at the design stage. We also had to match these programming rules to at least one of the software security design principles identified in the literature [13].

From this we found out that these secure coding rules can be basically measured as one or more of the object-oriented software design quality properties (e.g. encapsulation and cohesion). At first, we tried absolute metrics (e.g. the total number of classified attributes in a class). However, these were rejected due to the fact that they were not capable of comparing the relative security level of widely different designs. By instead using ratios we made the metrics directly comparable even with classes of different sizes.

The major rule we used to validate the metrics was that there must be no two designs which are different in terms of their encapsulation and cohesion but yield identical results for our security metrics. Different designs must have different security levels if their differences are relevant to secure coding rules or design principles.
7. Conclusion and Future Work

In this work, we have defined a number of security metrics for object-oriented designs. These metrics are easy to capture and apply once a given class is designed and annotated using UMLsec and SPARK's annotations. The metrics not only allow designers to define the most secure design but they can also give indications of where any potential vulnerability occurs. They differ from code level metrics as they are easier to capture and don't require the software to be implemented. We have also shown how to directly compare the metrics results for various alternative designs and thus help choose the design which best satisfies a certain security design principle. The defined approach can also make it easier for systems designers to choose which refactoring methods to use to satisfy a certain security design principle.

Future work will include a more general analysis of which refactoring methods can make certain classes more secure than others by making specific changes to the design. At the time of writing we are also defining another suite of security metrics which cover the entire design of a multi-class object-oriented design, including coupling, inheritance, and polymorphism.

8. Acknowledgments

We wish to thank the anonymous reviewers for their suggestions, and Ewan Tempero for his many helpful comments on this work.

Alshammari gratefully acknowledges the Ministry of Higher Education in Saudi Arabia for sponsoring his PhD studies. Fidge and Corney's contribution to this research was funded in part by the Australian Research Council and the Defence Signals Directorate through ARC-LP grant LP0776344.

9. References