Assessing The Impact of Refactoring on Security-Critical Object-Oriented Designs

Bandar Alshammari[1], Colin Fidge[2] and Diane Corney[3]

Faculty of Science and Technology
Queensland University of Technology
Brisbane, Australia.
Email: [1] b.alshammari@student.qut.edu.au [2] c.fidge@qut.edu.au [3] d.corney@qut.edu.au

***Abstract*—Refactoring focuses on improving the reusability, maintainability and performance of programs. However, the impact of refactoring on the security of a given program has received little attention. In this work, we focus on the design of object-oriented applications and use metrics to assess the impact of a number of standard refactoring rules on their security by evaluating the metrics before and after refactoring. This assessment tells us which refactoring steps can increase the security level of a given program from the point of view of potential information flow, allowing application designers to improve their system's security at an early stage.**

***Index Terms*—Object-orientation; Security; Metrics; Refactoring**

I. Introduction

Although there have been many studies into secure coding techniques [1] [2], a more efficient strategy is to assess program security at design time. We have previously proposed a set of metrics which allow designers to compare the security of various alternative designs for a given object-oriented program [3] [4], by quantifying potential information flow from 'classified' data values.

Refactoring rules [5] are a well-established way of restructuring an object-oriented system without changing its functional behaviour, but the effect of refactoring on program security is less clear. Here we use our software security metrics to assess the impact of a number of standard refactoring steps on program security. This is done by first measuring the security of a given program's design using the metrics [3] [4] and then measuring the security of other refactored designs of the same program.

II. Related Work and Research Problem

Refactoring is defined as "a change made to the internal structure of a program to make it easy to understand and cheap to modify without changing its observable behavior" [5]. Using refactoring to enhance a program's security has been considered in a number of studies. Maruyama and Tokoda [6] investigated how certain changes could affect the security characteristics of a given program with regard to access modifiers. Their work shows which refactoring rules could change a class's accessibility level and therefore change its security level. Other work by Maruyama aims to improve the overall security of a given program's code by identifying its code vulnerabilities and defining a set of secure refactoring rules [7]. Furthermore, Smith and Thober have identified a refactoring approach for critical systems [8]. This approach aims to refactor a program's code into two modules; a high-security and a low-security one. However, these previous approaches don't quantify the impact of changes on the overall security level of a given program. Furthermore, they require full source code implementations of the programs, which is inevitably less efficient than finding problems at design time.

Instead, in this paper we measure the impact of a set of refactoring steps on the security of a given program's *design* using our security design metrics [3] [4]. These metrics measure annotated UML class diagrams for a given or planned program. The UML class diagrams are annotated using UMLsec's annotations to identify confidential data [9] and SPARK's annotations to express the information flow relations between attributes, methods and classes [10].

III. Summary of Security Design Metrics

This section briefly summarises the security design metrics which we have used previously to compare the security level of program designs [3] [4] as an adjunct to other well-established metrics for assessing design or program complexity [11]. The metrics measure security from the perspective of information flow based on the security design principles of "reducing the size of the attack surface" [12] and "least privilege" [13] [14], using security annotations introduced by the designer. The metrics used in this paper are shown in Table I. Each is a ratio in the range 0 to 1, with lower values considered more secure. Absolute metrics such as the number of critical classes, classified methods and classified attributes are also helpful but are not covered in this paper.

The data encapsulation-based metrics (CIDA, CCDA and COA) assess the accessibility of classified attributes and methods [3]. Classified attributes are those with a UMLsec "secrecy" label and classified methods are those that interact with at least one such attribute, directly or indirectly, as revealed by SPARK "**derives** *value* **from** *attribute*" annotations. The cohesion-based metrics measure the potential flow of classified attributes' values to accessor and mutator methods, penalising designs with a large amount of classified flow [3].

Table I
SECURITY METRICS DEFINITIONS

Metric	Description	Definition
CIDA	The number of classified instance public attributes *CIPA* to the number of classified attributes *CA* in a design.	$CIDA(D) = \frac{\lvert CIPA\rvert}{\lvert CA\rvert}$
CCDA	The number of classified class public attributes *CCPA* to the number of classified attributes *CA* in a design.	$CCDA(D) = \frac{\lvert CCPA\rvert}{\lvert CA\rvert}$
COA	The number of classified public methods *CPM* to the number of classified methods *CM* in a design.	$COA(D) = \frac{\lvert CPM\rvert}{\lvert CM\rvert}$
CMAI	The number of mutators which may interact with classified attributes $\alpha(CA_i)$ to the possible maximum number of mutators *MM* which could interact with classified attributes *CA* in a design.	$CMAI(D) = \frac{\sum_{i=1}^{\lvert CA\rvert}\alpha(CA_i)}{\lvert MM\rvert\times\lvert CA\rvert}$
CAAI	The number of accessors which may interact with classified attributes $\beta(CA_i)$ to the possible maximum number of accessors *AM* which could interact with classified attributes *CA* in a design.	$CAAI(D) = \frac{\sum_{i=1}^{\lvert CA\rvert}\beta(CA_i)}{\lvert AM\rvert\times\lvert CA\rvert}$
CAIW	The number of all methods which may interact with classified attributes $\gamma(CA_i)$ to the number of all methods which could have access to all attributes $\delta(A_j)$ in a design.	$CAIW(D) = \frac{\sum_{i=1}^{\lvert CA\rvert}\gamma(CA_i)}{\sum_{j=1}^{\lvert A\rvert}\delta(A_j)}$
CMW	The number of classified methods *CM* to the total number of methods *M* in a design.	$CMW(D) = \frac{\lvert CM\rvert}{\lvert M\rvert}$
CCC	The number of classes' links with classified attributes $\varepsilon(CA_i)$ to the number of possible links of all classes *C* with classified attributes *CA* in a design.	$CCC(D) = \frac{\sum_{i=1}^{\lvert CA\rvert}\varepsilon(CA_i)}{(\lvert C\rvert-1)\times\lvert CA\rvert}$
CPCC	The number of critical composed-part classes *CP* to the total number of critical classes *CC* in a design.	$CPCC(D) = 1-\left(\frac{\lvert CP\rvert}{\lvert CC\rvert}\right)$
CCE	The number of the non-finalised critical classes *ECC* to the total number of critical classes *CC* in a design.	$CCE(D) = \frac{\lvert ECC\rvert}{\lvert CC\rvert}$
CME	The number of non-finalised classified methods *ECM* to the number of classified methods *CM* in a design.	$CME(D) = \frac{\lvert ECM\rvert}{\lvert CM\rvert}$
CSP	The number of critical superclasses *CSC* to the number of critical classes *CC* in an inheritance hierarchy.	$CSP(H) = \frac{\lvert CSC\rvert}{\lvert CC\rvert}$
CSI	The sum of classes which may inherit from each critical superclass $\varepsilon(CSC_k)$ to the number of possible inheritances from all classes *C* to all critical classes *CC* in an inheritance hierarchy.	$CSI(H) = \frac{\sum_{k=1}^{\lvert CSC\rvert}\varepsilon(CSC_k)}{(\lvert C\rvert-1)\times\lvert CC\rvert}$
CMI	The number of classified methods which can be inherited *MI* to the number of classified methods *CM* in an inheritance hierarchy.	$CMI(H) = \frac{\lvert MI\rvert}{\lvert CM\rvert}$
CAI	The number of classified attributes which can be inherited *AI* to the number of classified attributes *CA* in an inheritance hierarchy.	$CAI(H) = \frac{\lvert AI\rvert}{\lvert CA\rvert}$
CDP	The number of critical classes *CC* to the number of classes *C* in a design.	$CDP(D) = \frac{\lvert CC\rvert}{\lvert C\rvert}$

The coupling-based metric (CCC) measures interactions between classes and classified attributes, rewarding designs that minimise such interactions [4], because it is known that strong coupling makes security attacks easier [15]. 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 [4]. A class is considered "critical" if it contains a classified attribute or an attribute which derives its value from a classified one. The extensibility-based metrics (CCE and CME) reward designs with fewer opportunities for extending critical classes or classified methods since these are points at which an attacker can access classified data without affecting the system's observable behaviour [16]. The inheritance-based metrics (CSP, CSI, CMI and CAI) reward designs with fewer opportunities for inheriting from critical superclasses, since these allow subclasses to gain privileges over classified data [4]. Finally, the design size-based metric (CDP) rewards designs with a lower proportion of critical classes [4]. In situations where the design does not have the relevant constructs or features, which would produce a zero denominator in a metric, the whole metric is treated as zero.

IV. ASSESSMENT OF REFACTORING RULES

This section identifies the refactoring rules which we have determined may have an impact on the security of a given program. It also explains how these standard refactoring rules may affect the security design metrics defined in Section III.

A. Identifying Security-Critical Design Refactoring Rules

Table II lists standard refactoring rules [5] [17] which are applicable at the design stage and which may affect the security level of an object-oriented design. We distinguish their effect on classified and non-classified features as shown in Table III, and have studied their impact on confidential data accessibility and hence the overall security of that program. All of these rules may have an impact on the size of the design's 'attack surface' [12] and 'least privilege' [13] [14].

B. Assessing Security-Critical Design Refactoring Rules

In this section we analyse how the refactoring rules shown in Table III may affect the security design metrics defined in Section III.

For example, refactoring rules Encapsulate Classified Field and Hide Classified Method can improve security with regard to the Data Encapsulation-based metrics in two cases. One is

Table II
DESIGN REFACTORING RULES

Refactoring Rule	Effect
Encapsulate Field	Changes the access modifier of public fields to private.
Inline Field	Combines two fields or more into one if they are always used together.
Extract Field	Creates a new field from an existing one if its information can be used separately.
Pull Up Field	If two subclasses have the same field then this rule moves this field to their superclass.
Push Down Field	If a field is used by only some subclasses then this rule moves this field to those subclasses.
Move Field	Moves a field to another class.
Hide Method	Makes public methods private if not used by another class.
Inline Method	Combines two methods if they are always used together.
Extract Method	Creates a new method from an existing one.
Finalise Method	Declares a method as "final" to prevent it from being extended.
Pull Up Method	If two subclasses have the same method then this rule moves the method to their superclass.
Push Down Method	If a method is used by only some subclasses classes then this rule moves the method to those subclasses.
Move Method	Moves a method to another class.
Inline Class	Combines two classes if they are always used together.
Extract Class	Creates a new class from an existing one.
Finalise Class	Declares a class as "final" to prevent it from being extended.
Extract Superclass	If two subclasses have similar features, this rule creates a superclass and moves these features into it.
Extract Subclass	If two superclasses have similar features, this rule creates a subclass and moves these features into it.

if public classified fields have been encapsulated to be private using the Encapsulate Classified Field refactoring rule. This will make the program more secure in terms of the CIDA and CCDA metrics. Furthermore, when refactoring rule Hide Classified Method is applied to public classified methods to make them private, this will reduce the COA metric, making the program more secure in this regard.

Refactoring rules Inline Classified Field, Inline Classified Method, Extract Non-Classified Field and Extract Non-Classified Method could maintain or improve the security of the program with regard to the Cohesion-based security metrics (CMAI, CAAI, CAIW and CMW) in many cases. Using the Inline Classified Field and Inline Classified Method rules to inline classified attributes and classified methods will reduce the overall number of classified attributes and classified methods, and thus make the program more secure. Furthermore, using the Extract Non-Classified Field and Extract Non-Classified Method rules to separate non-classified attributes and methods from classified ones will decrease the proportion of classified attributes and methods, also making the program more secure.

Refactoring rules Extract Non-Critical Class, Extract Superclass, Extract Subclass and Move Classified Field can improve security with regard to the Coupling-based security metrics. Extract Non-Critical Class can be used to extract a non-critical class from an existing critical one, which increases the proportion of non-critical classes in the design and makes the critical ones simpler. The Move Classified Field rule can be used to move a classified field to a critical class which interacts with it. This will reduce the number of links with classified fields and thus reduce the CCC metric. Reducing this metric can also be achieved by introducing inheritance to related classes which use similar classified fields, which can be done by either Extract Critical Superclass or Extract Non-Critical Superclass and Extract Critical Subclass or Extract Non-Critical Subclass. This will change the coupling with classified attributes to be through inheritance which reduces this metric and makes the program more secure in this regard.

Refactoring rules Extract Composed-Part Critical Class, Move Classified Field and Move Classified Method can also make programs more secure in terms of Compositionality. They should be used in a particular way, to extract a composed-part critical class (i.e. an inner class of the outer class) using the Extract Composed-Part Critical Class rule. Then, the next step is to move the new class's related classified attributes and methods from the outer class using the Move Classified Field and Move Classified Method rules. This will increase the proportion of composed-part critical classes to the total number of critical classes, thus making the program more secure in terms of the CPCC metric.

The Extract Non-Critical Class, Move Non-Classified Field and Move Non-Classified Method rules can lower the proportion of critical classes to make programs more secure with regard to the Design Size metric (CDP), by extracting a non-critical class from an existing critical one. This will involve using the Move Non-Classified Field and Move Non-Classified Method rules to move the non-classified attributes and methods into the new class. Furthermore, refactoring rules Inline Critical Class, Move Classified Field and Move Classified Method can lower the proportion of critical classes to make programs more secure with regard to the Design Size metric (CDP), by combining two critical classes into one critical class. This will cause the relevant classified attributes and methods to be moved to their critical class using the Move Classified Field and Move Classified Method rules.

The Finalise Critical Class and Finalise Classified Method refactoring rules can make programs achieve a higher level of security in terms of Extensibility. Finalise Critical Class can be used to make critical classes 'final' to prevent other classes from extending them. This will increase the number of non-extendable critical classes, and thus reduces the CCE metric. The CME metric can be reduced by finalising classified methods, using the Finalise Classified Method rule.

A number of refactoring rules could allow subclasses to acquire more privileges over classified data, and decrease privileges of superclasses over such data, making programs more secure with regard to the Inheritance-based metrics. These rules include Extract Non-Critical Superclass, Extract

Table III
SECURITY-CRITICAL DESIGN REFACTORING RULES

Security Refactoring Rule	**Identifier**	**Effect**
Encapsulate Classified Field	RNCF	Changes the access modifier of classified public fields to private.
Encapsulate Non-Classified Field	RNNF	Changes the access modifier of non-classifed public fields to private.
Inline Classified Field	RICF	Combines two classified fields or more into one classified field if they are always used together.
Inline Non-Classified Field	RINF	Combines two classified and non-classified fields or more into one classified field if they are always used together.
Extract Classified Field	RECF	Creates a new classified field from an existing classified field if its information can be used separately.
Extract Non-Classified Field	RENF	Creates a new non-classified field from an existing classified or non-classified field if its information can be used separately.
Pull Up Classified Field	RPUCF	If two subclasses have the same classified field then this rule moves this field to their superclass.
Pull Up Non-Classified Field	RPUNF	If two subclasses have the same non-classified field then this rule moves this field to their superclass.
Push Down Classified Field	RPDCF	If a classified field is used by only some subclasses then this rule moves this field to those subclasses.
Push Down Non-Classified Field	RPDNF	If a non-classified field is used by only some subclasses then this rule moves this field to those subclasses.
Move Classified Field	RMCF	Moves a classified field to a critical class.
Move Non-Classified Field	RMNF	Moves a non-classified field to a critical class.
Hide Classified Method	RHCM	Makes classified public methods private if not used by another class.
Hide Non-Classified Method	RHNM	Makes non-classified public methods private if not used by another class.
Inline Classified Method	RICM	Combines two classified methods or more into one classified method if they are always used together.
Inline Non-Classified Method	RINM	Combines two classified and non-classified methods or more into one classified method if they are always used together.
Extract Classified Method	RECM	Creates a new classified method from an existing classified method if its information can be used separately.
Extract Non-Classified Method	RENM	Creates a new non-classified method from an existing classified or non-classified method if its information can be used separately.
Finalise Classified Method	RFCM	Declares a classified method as “final” to prevent it from being extended.
Finalise Non-Classified Method	RFNM	Declares a non-classified method as “final” to prevent it from being extended.
Pull Up Classified Method	RPUCM	If two subclasses have the same classified method then this rule moves this method to their superclass.
Pull Up Non-Classified Method	RPUNM	If two subclasses have the same non-classified method then this rule moves this method to their superclass.
Push Down Classified Method	RPDCM	If a classified method is used by only some subclasses then this rule moves this method to those subclasses.
Push Down Non-Classified Method	RPDNM	If a non-classified method is used by only some subclasses then this rule moves this method to those subclasses.
Move Classified Method	RMCM	Moves a classified method to a critical class.
Move Non-Classified Method	RMNM	Moves a non-classified method to a critical class.
Inline Critical Class	RICC	Combines two critical classes or more into one critical class if they are always used together.
Inline Non-Critical Class	RINC	Combines two critical and non-critical classes or more into one critical class if they are always used together.
Extract Critical Class	RECC	Creates a new critical class from an existing critical one.
Extract Composed-Part Critical Class	RECPCC	Creates a new composed-part critical class from an existing critical class.
Extract Non-Critical Class	RENC	Creates a new non-critical class from an existing critical one.
Finalise Critical Class	RFCC	Declares a critical class as “final” to prevent it from being extended.
Finalise Non-Critical Class	RFNC	Declares a non-critical class as “final” to prevent it from being extended.
Extract Critical Superclass	RECSP	If two critical subclasses have similar classified features, this rule creates a critical superclass and moves these features into it.
Extract Non-Critical Superclass	RENSP	If two critical subclasses have similar non-classified features, this rule creates a non-critical superclass and moves these features into it.
Extract Critical Subclass	RECSB	If two critical superclasses have similar classified features, this rule creates a critical subclass and moves these features into it.
Extract Non-Critical Subclass	RENSB	If two critical superclasses have similar non-classified features, this rule creates a non-critical subclass and moves these features into it.

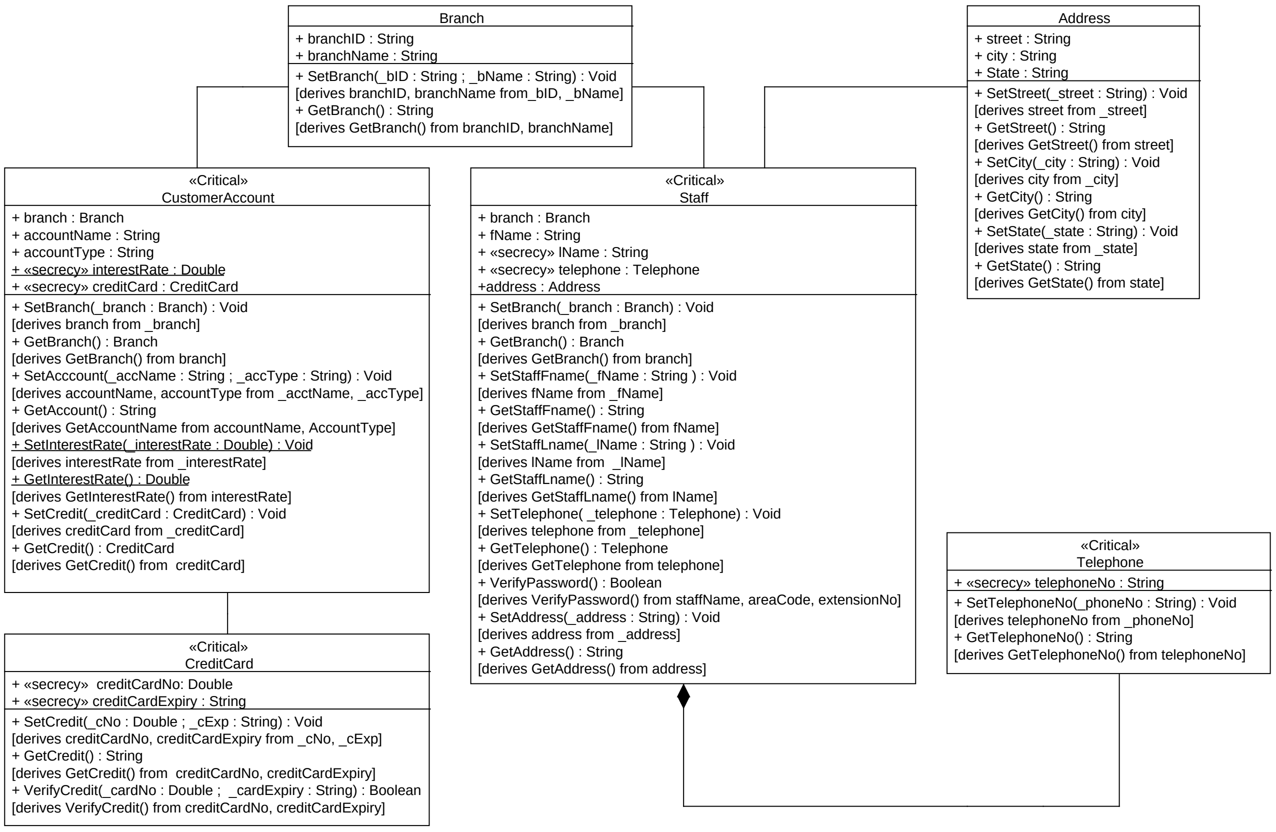
Branch
+ branchID : String
+ branchName : String
+ SetBranch(_bID : String ; _bName : String) : Void
[derives branchID, branchName from_bID, _bName]
+ GetBranch() : String
[derives GetBranch() from branchID, branchName]
Address
+ street : String
+ city : String
+ State : String
+ SetStreet(_street : String) : Void
[derives street from _street]
+ GetStreet() : String
[derives GetStreet() from street]
+ SetCity(_city : String) : Void
[derives city from _city]
+ GetCity() : String
[derives GetCity() from city]
+ SetState(_state : String) : Void
[derives state from _state]
+ GetState() : String
[derives GetState() from state]
«Critical»
CustomerAccount
+ branch : Branch
+ accountName : String
+ accountType : String
+ «secrecy» interestRate : Double
+ «secrecy» creditCard : CreditCard
+ SetBranch(_branch : Branch) : Void
[derives branch from _branch]
+ GetBranch() : Branch
[derives GetBranch() from branch]
+ SetAcccount(_accName : String ; _accType : String) : Void
[derives accountName, accountType from _acctName, _accType]
+ GetAccount() : String
[derives GetAccountName from accountName, AccountType]
+ SetInterestRate(_interestRate : Double) : Void
[derives interestRate from _interestRate]
+ GetInterestRate() : Double
[derives GetInterestRate() from interestRate]
+ SetCredit(_creditCard : CreditCard) : Void
[derives creditCard from _creditCard]
+ GetCredit() : CreditCard
[derives GetCredit() from creditCard]
«Critical»
Staff
+ branch : Branch
+ fName : String
+ «secrecy» lName : String
+ «secrecy» telephone : Telephone
+address : Address
+ SetBranch(_branch : Branch) : Void
[derives branch from _branch]
+ GetBranch() : Branch
[derives GetBranch() from branch]
+ SetStaffFname(_fName : String) : Void
[derives fName from _fName]
+ GetStaffFname() : String
[derives GetStaffFname() from fName]
+ SetStaffLname(_lName : String) : Void
[derives lName from _lName]
+ GetStaffLname() : String
[derives GetStaffLname() from lName]
+ SetTelephone(_telephone : Telephone) : Void
[derives telephone from _telephone]
+ GetTelephone() : Telephone
[derives GetTelephone from telephone]
+ VerifyPassword() : Boolean
[derives VerifyPassword() from staffName, areaCode, extensionNo]
+ SetAddress(_address : String) : Void
[derives address from _address]
+ GetAddress() : String
[derives GetAddress() from address]
«Critical»
Telephone
+ «secrecy» telephoneNo : String
+ SetTelephoneNo(_phoneNo : String) : Void
[derives telephoneNo from _phoneNo]
+ GetTelephoneNo() : String
[derives GetTelephoneNo() from telephoneNo]
«Critical»
CreditCard
+ «secrecy» creditCardNo: Double
+ «secrecy» creditCardExpiry : String
+ SetCredit(_cNo : Double ; _cExp : String) : Void
[derives creditCardNo, creditCardExpiry from _cNo, _cExp]
+ GetCredit() : String
[derives GetCredit() from creditCardNo, creditCardExpiry]
+ VerifyCredit(_cardNo : Double ; _cardExpiry : String) : Boolean
[derives VerifyCredit() from creditCardNo, creditCardExpiry]

Figure 1. Bank Account Hierarchy 1

Critical Subclass, Pull Up Non-Classified Field, Pull Up Non-Classified Method, Push Down Classified Field and Push Down Classified Method. This can be achieved through extracting a non-critical superclass for similar non-classified features, and then extracting critical subclasses for their classified data using the Extract Non-Critical Superclass and Extract Critical Subclass rules. This will reduce the number of critical superclasses in a design, and hence reduce the CSP metric. Moreover, it will reduce the number of critical superclasses that could be inherited, which reduces the CSI metric. Using the Pull Up Non-Classified Field and Pull Up Non-Classified Method rules to move non-classified attributes and methods to the non-critical superclasses in addition to using the Push Down Classified Field and Push Down Classified Method rules to move classified attributes and methods to the critical subclasses will reduce the number of classified attributes and classified methods which could be inherited. This will thus reduce the CAI and CMI metrics.

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.

V. Case Study

The following case study illustrates how applying the refactoring rules shown in Table III impacts the security of a design in a way measurable by our metrics in Table I, as predicted in Section IV. In order for this assessment to take place, a complete annotated UML class diagram is required. It must include UMLsec and SPARK annotations in addition to the standard elements of a class diagram in order to identify classified data items and their uses.

A. Original Annotated Design

The class diagram in Figure 1 has been annotated using UMLsec's and SPARK's annotations. The Bank Account system class diagram is responsible for storing information about customer accounts and bank staff who belong to a certain branch. The Branch class contains the branch ID and name. In class Customer Account, a bank account can be either a savings or a credit account which is determined by the Account Type attribute. The value of a Savings account's interest rate is different from a Credit account's but it is a class (static) attribute since its value is shared for all objects of the initialised class. It is underlined in Figure 1 to be consistent with the UML class diagram rules. The Customer Account class also stores attributes of both the savings and credit accounts. The Credit Card class stores the account's credit card number and expiry date. We assume that the account's interest rate, credit card number and expiry date attributes are sensitive and are meant to be kept secret.

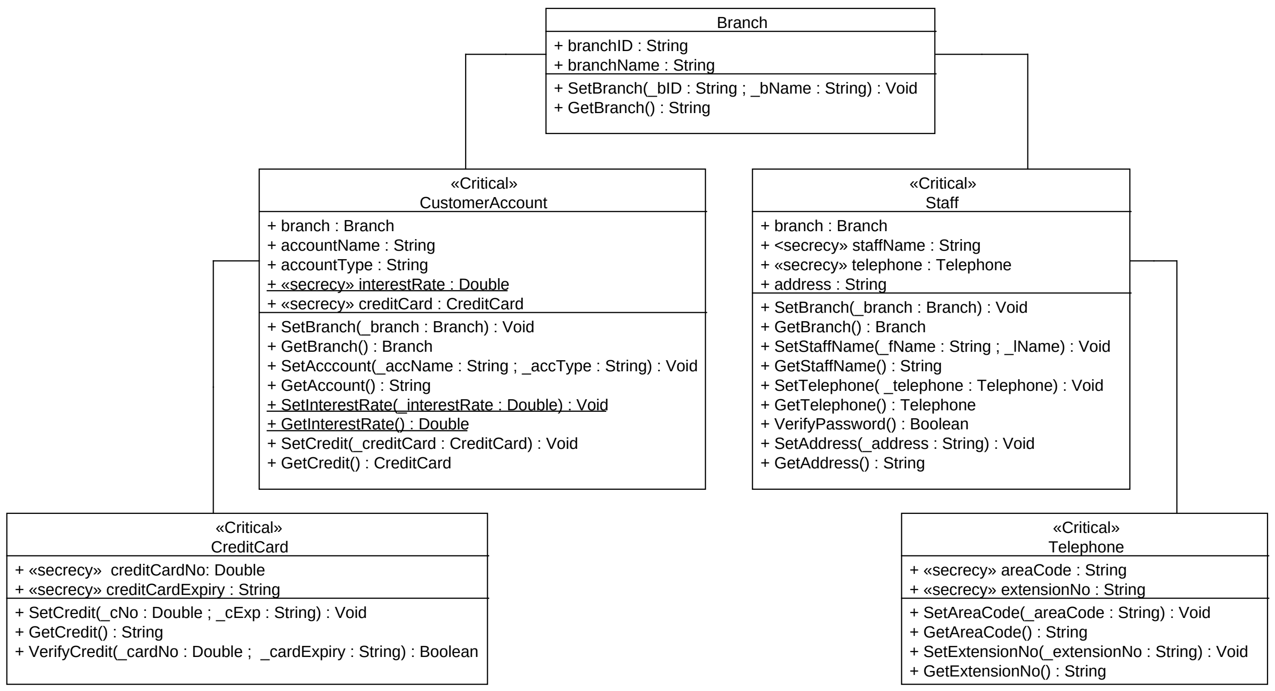
Branch
+ branchID : String
+ branchName : String
+ SetBranch(_bID : String ; _bName : String) : Void
+ GetBranch() : String
«Critical»
CustomerAccount
+ branch : Branch
+ accountName : String
+ accountType : String
+ «secrecy» interestRate : Double
+ «secrecy» creditCard : CreditCard
+ SetBranch(_branch : Branch) : Void
+ GetBranch() : Branch
+ SetAcccount(_accName : String ; _accType : String) : Void
+ GetAccount() : String
+ SetInterestRate(_interestRate : Double) : Void
+ GetInterestRate() : Double
+ SetCredit(_creditCard : CreditCard) : Void
+ GetCredit() : CreditCard
«Critical»
Staff
+ branch : Branch
+ <secrecy» staffName : String
+ «secrecy» telephone : Telephone
+ address : String
+ SetBranch(_branch : Branch) : Void
+ GetBranch() : Branch
+ SetStaffName(_fName : String ; _lName) : Void
+ GetStaffName() : String
+ SetTelephone(_telephone : Telephone) : Void
+ GetTelephone() : Telephone
+ VerifyPassword() : Boolean
+ SetAddress(_address : String) : Void
+ GetAddress() : String
«Critical»
CreditCard
+ «secrecy» creditCardNo: Double
+ «secrecy» creditCardExpiry : String
+ SetCredit(_cNo : Double ; _cExp : String) : Void
+ GetCredit() : String
+ VerifyCredit(_cardNo : Double ; _cardExpiry : String) : Boolean
«Critical»
Telephone
+ «secrecy» areaCode : String
+ «secrecy» extensionNo : String
+ SetAreaCode(_areaCode : String) : Void
+ GetAreaCode() : String
+ SetExtensionNo(_extensionNo : String) : Void
+ GetExtensionNo() : String

Figure 2. Bank Account Hierarchy 2

The Staff class is responsible for storing information about bank staff. This information consists of the branch where the staff work, the staff member's name (first and last names) and address details (i.e. street, city and state) which are stored in the Address class. The Telephone class is responsible for storing information about staff member's area code and internal phone extension, which is intended for use only within the organisation and should be kept secret to prevent direct calls from bank customers (who should call via the switchboard). Additionally, we assume that the staff member's last name also needs to be kept secret since it is used as a user name for the bank's computer system. All of these classes contain operations which are responsible for mutating and accessing these details once they have been requested. The various 'derives … from … ' annotations tell us how attributes, method parameters and return values are related.

B. Refactored Designs

Figures 2 and 3 show two refactored versions of the original design using some of the refactoring rules in Table III. (The two additional designs omit the SPARK "derives … from … " annotations for brevity.) The two refactorings aim to make the original design clearer and more maintainable, respectively, and are typical of the kinds of changes that a designer might reasonably contemplate.

For instance, Figure 2 differs from the original design in the number of classes, which is now five instead of six. This was done to improve the understandability of the program [11], using Inline Non-Critical Class to inline the non-critical Address class with the critical Staff class. This was followed by Move Non-Classified Field and Move Non-Classified Method to move all the relevant non-classified attributes and methods from the pre-exisiting Address class to the Staff class. However, these steps make the program less secure with regard to its Design Size, and thus reduce the CDP metric. These rules also reduce the proportion of non-critical classes over critical ones, increasing the CCC metric. The new design has also used three other refactoring rules which increase the composition metric (CPCC). Extract Critical Class was used to change the existing composed-part critical Telephone class into an independent critical class, and Move Classified Field and Move Classified Method were used to move the Telephone class's classified fields and methods inside it. A number of rules have also been used which made the design less secure in terms of the Cohesion-based metrics including Inline Non-Classified Field, Inline Non-Classified Method, Extract Classified Field and Extract Classified Method. The Inline Non-Classified Field rule was used to combine one classified and one non-classified attribute into one classified attribute, i.e. the first name and last name from the Staff class have been combined into one classified staff name attribute. This has resulted in using the Inline Non-Classified Method to merge previous methods of the two inlined classified and non-classified attributes into one classified method. Finally, Extract Classified Field has been used to extract two classified attributes, i.e. the area code and extension number, of the one classified attribute telephone number in the Telephone class. This, of course, has resulted in using the Extract Classified Method rule to extract classified methods which mutate and access the new classified attributes

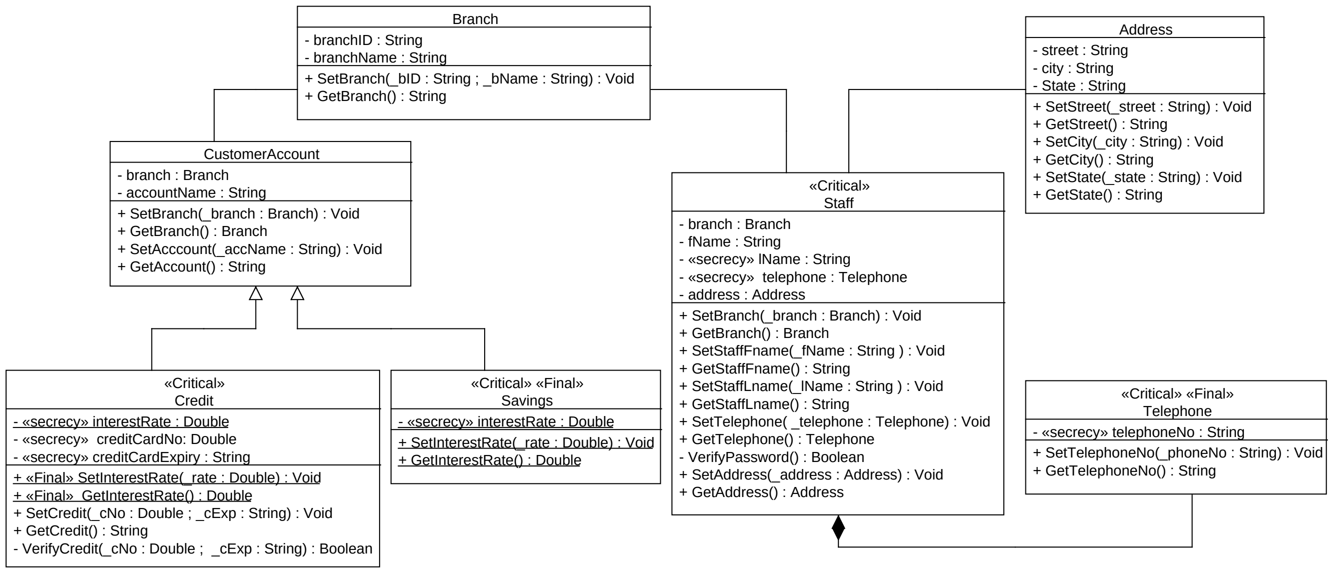
Branch
- branchID : String
- branchName : String
+ SetBranch(_bID : String ; _bName : String) : Void
+ GetBranch() : String
Address
- street : String
- city : String
- State : String
+ SetStreet(_street : String) : Void
+ GetStreet() : String
+ SetCity(_city : String) : Void
+ GetCity() : String
+ SetState(_state : String) : Void
+ GetState() : String
CustomerAccount
- branch : Branch
- accountName : String
+ SetBranch(_branch : Branch) : Void
+ GetBranch() : Branch
+ SetAcccount(_accName : String) : Void
+ GetAccount() : String
«Critical»
Staff
- branch : Branch
- fName : String
- «secrecy» lName : String
- «secrecy» telephone : Telephone
- address : Address
+ SetBranch(_branch : Branch) : Void
+ GetBranch() : Branch
+ SetStaffFname(_fName : String) : Void
+ GetStaffFname() : String
+ SetStaffLname(_lName : String) : Void
+ GetStaffLname() : String
+ SetTelephone(_telephone : Telephone) : Void
+ GetTelephone() : Telephone
- VerifyPassword() : Boolean
+ SetAddress(_address : Address) : Void
+ GetAddress() : Address
«Critical»
Credit
- «secrecy» interestRate : Double
- «secrecy» creditCardNo: Double
- «secrecy» creditCardExpiry : String
+ «Final» SetInterestRate(_rate : Double) : Void
+ «Final» GetInterestRate() : Double
+ SetCredit(_cNo : Double ; _cExp : String) : Void
+ GetCredit() : String
- VerifyCredit(_cNo : Double ; _cExp : String) : Boolean
«Critical» «Final»
Savings
- «secrecy» interestRate : Double
+ SetInterestRate(_rate : Double) : Void
+ GetInterestRate() : Double
«Critical» «Final»
Telephone
- «secrecy» telephoneNo : String
+ SetTelephoneNo(_phoneNo : String) : Void
+ GetTelephoneNo() : String

Figure 3. Bank Account Hierarchy 3

of the Telephone area code and extension number. In summary, therefore, we expect our metrics to show that this new design is less secure than the original one.

Another refactored design is shown in Figure 3. It has been refactored from the original design in Figure 1 but it has used inheritance for the CustomerAccount and CreditCard classes for extendability and effectiveness reasons [11]. This has resulted in creating three classes: CustomerAccount, Savings and Credit. The Extract Non-Critical Superclass rule was used to extract a non-critical superclass with the shared non-classified attributes and methods for both of the new subclasses: Savings and Credit. Since these two classes have different classified attributes and methods, this has resulted in extracting them as critical subclasses using the Extract Critical Subclass rule. These two refactoring rules reduce the inheritance metrics of CSP and CSI. This has caused the shared non-classified attributes and methods to be pulled up to their non-critical superclass using the Pull Up Non-Classified Field and Pull Up Non-Classified Method rules. Similarly, different classified attributes and methods for both classes have been pushed down to their relevant subclasses via the Push Down Classified Field and Push Down Classified Method rules. These rules, however, reduce the inheritance metrics of CMI and CAI. Furthermore, the new design has managed to reduce the proportion of public classified instance and class attributes and also public classified methods by using the Encapsulate Classified Field and Hide Classified Method rules. Such rules reduce the Data Encapsulation metrics CIDA, CCDA and COA. This has resulted in making the public classified instance and class attributes in all classes private. This has also changed the modifier access of two classified methods to be private (i.e. VerifyCredit in the CreditCard class and VerifyPassword in the Staff class since they were only used within their own class). The design has used Finalise Critical Class to make the critical classes of Telephone and Savings final to prevent these classes from being extended by adversaries. Similarly, it has also used Finalise Classified Method to make classified methods SetInterestRate and GetInterestRate in the Credit class final. These rules make the design more secure in terms of the Extensibility (CCE and CME) metrics. Overall, therefore, we expect these steps to result in the most secure design of all.

C. *Security Metrics Results*

Tables IV and V show the results of applying the security metrics introduced in Section III to the three designs shown in Figures 1, 2 and 3.

Given that lower values of each metric are considered more secure, it can be seen that the results of these different designs vary with regard to their security level for many of the metrics. In general, the more refactoring rules that were used to refactor a given design according to the defined cases in Section IV-B, the more design-level security metrics are affected. Overall, Account 3 shows the most secure design with regard to all of the metrics. By contrast, Account 2 shows the least secure design in terms of these metrics.

In comparison, Account 1 is more secure than Account 2 in most of the security-relevant metrics including the Cohesion, Composition, Coupling, and Design Size-based metrics. Nevertheless, it shows the same results in terms of the Data Encapsulation, Extensibility and Inheritance-based metrics, because none of the changes affect the accessibility of classified attributes and methods, the position of critical classes in the hierarchy or the extensibility of classes. In this case, therefore, our refactoring steps made the design's security

Table IV
RESULTS FOR INDIVIDUAL CLASS SECURITY METRICS

Design	CIDA	CCDA	COA	CMAI	CAAI	CAIW	CMW
Account 1	0	0	0.857	0.067	0.092	0.450	0.437
Account 2	1	1	1	0.083	0.107	0.588	0.615
Account 3	0	0	0.857	0.067	0.092	0.444	0.437

Table V
RESULTS FOR MULTI-CLASS SECURITY METRICS

Design	CPCC	CCC	CCE	CME	CSP	CSI	CMI	CAI	CDP
Account 1	0.75	0.057	1	1	0	0	0	0	0.67
Account 2	1	0.062	1	1	0	0	0	0	0.80
Account 3	0.75	0.024	0.60	0.43	0	0	0	0	0.57

measurably worse.

Furthermore, the security refactoring rules applied to Account 3 have improved the security level of other metrics including the Coupling and Design Size-based ones. Account 3 differs from Account 1 by changing the access of all classified attributes and a number of classified methods to be private. It has also declared a number of critical classes and classified methods as "final" to prevent their extension. Additionally, it has used inheritance in such a way that there exists neither critical superclasses nor classified attributes or methods which could be inherited.

D. Security-Critical Design Refactoring Rules Assessment

Based on the previous case study, we have identified choices of refactoring rules which could make the security of a given design either more secure, less secure or maintain the security of a given design as shown in Table VI. This list shows all of the security-critical refactoring rules in Table III and their expected impact on security for each of the security metrics identified in Table I. Their impact is shown with regard to each security metric in four different ways: "$\uparrow$" means may *increase* that security metric, "$\downarrow$" means may *decrease* that security metric, "$\updownarrow$" means may *increase* or *decrease* that security metric and "$-$" means has no impact on that metric. The final column of the table indicates how the refactoring rules affect the security of a design based on a sum of the previous columns. This assessment must be interpreted with some caution, however, since it places no particular weight on each of the metrics. Whether or not all the metrics should be viewed as equally valuable depends very much on the particular designer's goals and motivations. Table VI shows that there are twenty refactoring rules which make security better, twelve rules which make security worse and the remaining four rules have no impact on security overall.

VI. CONCLUSIONS AND FUTURE WORK

In this work, we have assessed how refactoring steps can influence the security of a given object-oriented design as measured by a set of security metrics. The results show that refactoring rules can improve the overall security of a given program in a quantifiable way when used to restrict the accessibility of classified data. In particular, our assessment has shown that there are twenty different refactoring rules which can improve the overall security of a given design.

Future work will include: defining a set of refactoring rules specifically designed to improve security properties, by ensuring that their application always improves or maintains particular security metrics; development of a tool for automatically calculating the metrics and summarising them in an easy-to-understand form; and gaining practical experience with applying the metrics in order to identify the most useful or helpful ones.

ACKNOWLEDGMENTS

We wish to thank the anonymous reviewers for their helpful comments. 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.

REFERENCES

[1] G. McGraw, *Software Security: Building Security In*. Upper Saddle River, NJ: Addison-Wesley, 2006.

[2] M. Howard and D. LeBlanc, *Writing Secure Code*. Redmond, Wash.: Microsoft Press, 2002.

[3] B. Alshammari, C. J. Fidge, and D. Corney, "Security metrics for object-oriented class designs," in *Proceedings of the Ninth International Conference on Quality Software (QSIC 2009)*, (Jeju, Korea), pp. 11–20, IEEE Computer Society, 2009.

[4] B. Alshammari, C. J. Fidge, and D. Corney, "Security metrics for object-oriented designs," in *Proceedings of the Twenty-First Australian Software Engineering Conference (ASWEC 2010), Auckland, 6–9 April* (J. Noble and C. J. Fidge, eds.), (California, USA), pp. 55–64, IEEE Computer Society, 2010.

[5] M. Fowler, *Refactoring: Improving The Design of Existing Code*. Reading, MA: Addison-Wesley, 1999.

[6] K. Maruyama and K. Tokoda, "Security-aware refactoring alerting its impact on code vulnerabilities," in *Proceedings of the 15th Asia-Pacific Software Engineering Conference (APSEC 2008)*, IEEE Computer Society, 2008. 1488052 445-452.

[7] K. Maruyama, "Secure refactoring - improving the security level of existing code," in *Proceedings of the Second International Conference on Software and Data Technologies (ICSOFT 2007)*, (Barcelona, Spain), pp. 222–229, 2007.

[8] S. F. Smith and M. Thober, "Refactoring programs to secure information flows," in *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, (Ontario, Canada), ACM, 2006. 1134758 75-84.

[9] J. Jürjens, *Secure systems development with UML*. Berlin, Germany: Springer, 2005.

Table VI
SECURITY-CRITICAL DESIGN REFACTORING RULES ASSESSMENT

Rule	CIDA	CCDA	COA	CMAI	CAAI	CAIW	CMW	CCC	CPCC	CCE	CME	CSP	CSI	CMI	CAI	CDP	Total
RNCF	↓	↓	–	–	–	–	–	↓	–	–	–	–	–	–	–	–	↓
RNNF	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
RICF	–	–	–	↓	↓	↓	↓	–	–	–	–	–	–	–	–	–	↓
RINF	–	–	–	↑	↑	↑	↑	–	–	–	–	–	–	–	–	–	↑
RECF	–	–	–	↑	↑	↑	↑	–	–	–	–	–	–	–	–	–	↑
RENF	–	–	–	↓	↓	↓	↓	–	–	–	–	–	–	–	–	–	↓
RPUCF	–	–	–	–	–	↑	–	↑	–	–	–	↑	↑	–	↑	↑	↑
RPUNF	–	–	–	–	–	↓	–	↓	–	–	–	↓	↓	–	↓	–	↓
RPDCF	–	–	–	–	–	↓	–	↓	–	–	–	↓	↓	–	↓	↓	↓
RPDNF	–	–	–	–	–	↑	–	–	–	–	–	↑	↑	–	↑	–	↑
RMCF	–	–	–	↓	↓	↓	↓	↓	↓	–	–	–	–	–	–	↓	↓
RMNF	–	–	–	↑	↑	↑	↑	↑	–	–	–	–	–	–	–	↓	↑
RHCM	–	–	↓	–	–	–	–	↓	–	–	–	–	–	–	–	–	↓
RHNM	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
RICM	–	–	–	↓	↓	↓	↓	↓	–	–	–	–	–	–	–	–	↓
RINM	–	–	–	↑	↑	↑	↑	↑	–	–	–	–	–	–	–	–	↑
RECM	–	–	–	↑	↑	↑	↑	–	–	–	–	–	–	–	–	–	↑
RENM	–	–	–	↓	↓	↓	↓	↓	–	–	–	–	–	–	–	–	↓
RFCM	–	–	–	–	–	–	–	–	–	–	↓	–	–	–	–	–	↓
RFNM	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
RPUCM	–	–	–	–	–	↑	–	↑	–	–	–	↑	↑	↑	–	↑	↑
RPUNM	–	–	–	–	–	↓	–	↓	–	–	–	↓	↓	↓	–	–	↓
RPDCM	–	–	–	–	–	↓	–	↓	–	–	–	↓	↓	↓	–	↓	↓
RPDNM	–	–	–	–	–	↑	–	↑	–	–	–	↑	↑	↑	–	–	↑
RMCM	–	–	–	↓	↓	↓	↓	↓	↓	–	–	–	–	–	–	↓	↓
RMNM	–	–	–	↑	↑	↑	↑	↑	–	–	–	–	–	–	–	↓	↑
RICC	–	–	–	↓	↓	↓	↓	↕	↓	–	–	–	–	–	–	↓	↓
RINC	–	–	–	↑	↑	↑	↑	↓	–	–	–	–	–	–	–	↑	↑
RECC	–	–	–	–	–	–	–	↕	↑	–	–	–	–	–	–	↑	↑
RECPCC	–	–	–	–	–	–	–	↓	↓	–	–	–	–	–	–	↑	↓
RENC	–	–	–	↓	↓	↓	↓	↓	–	–	–	–	–	–	–	↓	↓
RFCC	–	–	–	–	–	–	–	–	–	↓	–	–	–	–	–	–	↓
RFNC	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
RECSP	–	–	–	–	–	–	–	↓	–	–	–	↑	↑	–	–	↑	↑
RENSP	–	–	–	–	–	–	–	↓	–	–	–	↓	↓	–	–	↓	↓
RECSB	–	–	–	–	–	–	–	↓	–	–	–	↓	↓	–	–	↑	↓
RENSB	–	–	–	–	–	–	–	↓	–	–	–	–	–	–	–	↓	↓

[10] J. Barnes, *High Integrity Software: The SPARK Approach to Safety and Security*. London, Great Britain: Addison-Wesley, 2003.

[11] J. Bansiya and C. G. Davis, “A hierarchical model for object-oriented design quality assessment,” *IEEE Transactions on Software Engineering*, vol. 28, pp. 4–17, 2002.

[12] M. Howard, “Attack surface: Mitigate security risks by minimizing the code you expose to untrusted users,” *Microsoft MSDN Magazine*, vol. 11, 2004.

[13] J. H. Saltzer and M. D. Schroeder, “The protection of information in operating systems,” in *Proceedings of the IEEE*, vol. 63, pp. 1278–1308, 1975.

[14] M. Bishop, *Computer Security: Art and Science*. Boston: Addison-Wesley, 2003.

[15] M. Y. Liu and I. Traore, “Empirical relation between coupling and attackability in software systems: a case study on DOS,” in *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security, Ottawa*, (Ottawa, Ontario, Canada), pp. 57–64, ACM, 2006.

[16] G. McGraw and E. Felten, *Securing Java: Getting Down to Business with Mobile Code*. New York: Wiley Computer Pub., second ed., 1999.

[17] M. Fowler et al., “Refactoring home page.” Retrieved July 9, 2010.