Quality Metrics for Assessing Security-Critical Computer Programs

By

Bandar M. Alshammari
Bachelor of Information Technology (University of Canberra, 2006)
Master of Computer and Communications Engineering (Queensland University of Technology, 2007)

A THESIS SUBMITTED TO
THE FACULTY OF SCIENCE AND TECHNOLOGY
OF QUEENSLAND UNIVERSITY OF TECHNOLOGY
IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Supervisors:
Prof. Colin Fidge (Principal)
Dr. Diane Corney (Associate)

Computer Science
Faculty of Science and Technology
Queensland University of Technology

December 2011
I dedicate this thesis to
the memory of my mother,

my father,

my wife,

and my son.
Keywords

Abstract

Existing secure software development principles tend to focus on coding vulnerabilities, such as buffer or integer overflows, that apply to individual program statements, or issues associated with the run-time environment, such as component isolation. Here we instead consider software security from the perspective of potential information flow through a program’s object-oriented module structure. In particular, we define a set of quantifiable “security metrics” which allow programmers to quickly and easily assess the overall security of a given source code program or object-oriented design.

Although measuring quality attributes of object-oriented programs for properties such as maintainability and performance has been well-covered in the literature, metrics which measure the quality of information security have received little attention. Moreover, existing security-relevant metrics assess a system either at a very high level, i.e., the whole system, or at a fine level of granularity, i.e., with respect to individual statements. These approaches make it hard and expensive to recognise a secure system from an early stage of development.

Instead, our security metrics are based on well-established compositional properties of object-oriented programs (i.e., data encapsulation, cohesion, coupling, composition, extensibility, inheritance and design size), combined with data flow analysis principles that trace potential information flow between high- and low-security system variables.

We first define a set of metrics to assess the security quality of a given object-oriented system based on its design artifacts, allowing defects to be detected at an early stage of development. We then extend these metrics to produce a second set applicable to object-oriented program source code. The resulting metrics make it easy to compare the relative security of functionally-equivalent system designs or source code programs so that, for instance, the security of two
different revisions of the same system can be compared directly.

This capability is further used to study the impact of specific refactoring rules on system security more generally, at both the design and code levels. By measuring the relative security of various programs refactored using different rules, we thus provide guidelines for the safe application of refactoring steps to security-critical programs.

Finally, to make it easy and efficient to measure a system design or program’s security, we have also developed a stand-alone software tool which automatically analyses and measures the security of UML designs and Java program code. The tool’s capabilities are demonstrated by applying it to a number of security-critical system designs and Java programs. Notably, the validity of the metrics is demonstrated empirically through measurements that confirm our expectation that program security typically improves as bugs are fixed, but worsens as new functionality is added.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keywords</td>
<td>v</td>
</tr>
<tr>
<td>Abstract</td>
<td>vii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xx</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xxii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>xxv</td>
</tr>
</tbody>
</table>

## INTRODUCTION

1 Introduction

1.1 Motivation | 4 |
1.2 Research Problem | 5 |
1.3 Research Questions | 5 |
1.4 Research Objectives and Approach | 6 |
1.4.1 Type Inference Model | 6 |
1.4.2 Annotations for Identifying Program’s Security-Critical Features | 7 |
1.4.3 Metrics for Quantifying the Security of Object-Oriented Designs | 8 |
1.4.4 Metrics for Quantifying the Security of Object-Oriented Programs | 8 |
1.4.5 Transformation Rules for Developing or Maintaining Secure Programs 8
1.4.6 Tool Support for Assessing Security-Critical Programs and Designs 9

1.5 Research Outcomes .................................................. 9
1.5.1 Type System .......................................................... 10
1.5.2 Security Annotations .............................................. 10
1.5.3 Security Measurements .......................................... 10
1.5.4 Secure Refactoring Rules ....................................... 11
1.5.5 A Security Assessment Tool ................................... 11

1.6 Thesis Outline .......................................................... 11

2 Literature Review ......................................................... 15
2.1 Introduction ............................................................. 15
2.2 Software Security Design Principles ............................... 18
2.2.1 Principle 1: Architecture Matches Program ................. 19
2.2.2 Principle 2: Secure the Weakest Link ....................... 19
2.2.3 Principle 3: Least Privilege .................................... 20
2.2.4 Principle 4: Fail Safe Defaults ................................. 21
2.2.5 Principle 5: Complete Mediation ............................. 22
2.2.6 Principle 6: Economy of Mechanism ....................... 22
2.2.7 Principle 7: Open Design ....................................... 23
2.2.8 Principle 8: Separation of Privileges ....................... 23
2.2.9 Principle 9: Least Common Mechanism ..................... 24
2.2.10 Principle 10: Psychological Acceptability ................. 24
2.2.11 Principle 11: Work Factor .................................... 25
2.2.12 Principle 12: Compromise Record ......................... 25
2.2.13 Principle 13: Practice Defence in Depth .................. 26
2.2.14 Principle 14: Fail Securely .................................... 26
2.2.15 Principle 15: Compartmentalize/Isolation ................ 27

x
2.2.16 Principle 16: Reduce the Size of the Attack Surface . . . . . . . . . . 28
2.2.17 Summary of Relevant Design Principles . . . . . . . . . . . . . . . . 29
2.3 Software Quality Measurements . . . . . . . . . . . . . . . . . . . . . . . 30
2.3.1 Software Security Metrics . . . . . . . . . . . . . . . . . . . . . . . 31
2.3.2 Validation of Software Metrics . . . . . . . . . . . . . . . . . . . . . . 34
2.3.3 Summary of Previous Security Metrics . . . . . . . . . . . . . . . . . 35
2.4 Refactoring . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
2.4.1 Refactoring Objectives . . . . . . . . . . . . . . . . . . . . . . . . . 36
2.4.2 The Refactoring activities . . . . . . . . . . . . . . . . . . . . . . . . 37
2.4.3 Refactoring and Performance . . . . . . . . . . . . . . . . . . . . . . 39
2.4.4 Refactoring and Design Quality . . . . . . . . . . . . . . . . . . . . . 40
2.4.5 Refactoring and Security . . . . . . . . . . . . . . . . . . . . . . . . 41
2.4.6 Significance of Security Metrics for Refactoring . . . . . . . . . . . . 42
2.5 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

II OBJECT-ORIENTED DESIGN-LEVEL SECURITY ASSESSMENT

3 Security Metrics for Object-Oriented Class Designs . . . . . . . . . . . . 47
3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
3.2 Assumptions and Annotations . . . . . . . . . . . . . . . . . . . . . . . . 49
3.3 Relevant Security Design Principles . . . . . . . . . . . . . . . . . . . . . 50
3.4 Security Design Metrics . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
3.4.1 Security Accessibility Metrics . . . . . . . . . . . . . . . . . . . . . . 52
3.4.2 Security Interactions Metrics . . . . . . . . . . . . . . . . . . . . . . 54
3.5 Individual-Class Design Metrics Case Study . . . . . . . . . . . . . . . . . 57
3.5.1 UML Class Diagrams . . . . . . . . . . . . . . . . . . . . . . . . . . 58
3.5.2 Metrics Results and Representations . . . . . . . . . . . . . . . . . . 62
III  OBJECT-ORIENTED CODE-LEVEL SECURITY ASSESSMENT

6  Security Metrics for Object-Oriented Programs

6.1  Introduction

6.2  Program Code Security Metrics Definitions

   6.2.1  Data Encapsulation-Based Security Metrics

   6.2.2  Cohesion-Based Security Metrics

   6.2.3  Coupling-Based Security Metric

   6.2.4  Composition-Based Security Metric

   6.2.5  Extensibility-Based Security Metrics

   6.2.6  Design Size-Based Security Metrics

   6.2.7  Inheritance-Based Security Metrics

6.3  Program Code Security Metrics Experimental Results

   6.3.1  Approach

   6.3.2  Programs Analysed

   6.3.3  Program Annotations

   6.3.4  Program Characteristics

   6.3.5  Programs Security Metrics Results

6.4  Conclusion

7  A Hierarchical Security Assessment Model for Object-Oriented Programs

7.1  Introduction

7.2  The Security Assessment Model

   7.2.1  Metrics for the Potential Flow of Classified Data

   7.2.2  Metrics for the Readability and Writability of Classified Data
7.2.3 Metrics for Specific Security Design Principles ........................................ 135
7.2.4 A Total Security Index ........................................................................ 136

7.3 Experimental Results ............................................................................ 137
7.3.1 Approach .......................................................................................... 138
7.3.2 The Programs Analysed .................................................................... 138
7.3.3 Program Annotations ......................................................................... 139
7.3.4 Analysis of Experimental Results ...................................................... 139

7.4 Conclusion ............................................................................................. 145

8 Security Assessment of Code-Level Refactoring Rules ......................... 147
8.1 Introduction ........................................................................................... 147
8.2 Security Assessment of Refactoring Rules ............................................ 148
8.3 Case Study ............................................................................................ 152
8.3.1 Original Program ............................................................................... 152
8.3.2 Refactored Programs ......................................................................... 154
8.3.3 Security Metrics Results ................................................................... 158
8.3.4 Code Refactoring Rules Assessment ............................................... 163
8.4 Conclusion ............................................................................................. 164

IV AUTOMATED TOOL FOR SECURITY EVALUATION ............................. 167

9 A Type Inference System for Defining Secure Data Flow ......................... 169
9.1 Introduction ........................................................................................... 169
9.2 A Security Type System ......................................................................... 170
9.2.1 Type Inference Rules Contexts ......................................................... 171
9.2.2 Inference Typing Rules ..................................................................... 173
9.2.3 Inference Subtyping Rules .............................................................. 176
9.3 Examples of Typing Security-Critical Code ........................................ 178
List of Figures

1.1 Thesis Structure ................................................................. 12
2.1 Fail Securely Example .......................................................... 27
2.2 The refactoring process ......................................................... 37
3.1 ContactNos 1 class diagram .................................................. 58
3.2 ContactNos 2 class diagram .................................................. 59
3.3 ContactNos 3 class diagram .................................................. 60
3.4 ContactNos 4 class diagram .................................................. 61
3.5 ContactNos 5 class diagram .................................................. 61
3.6 ContactNos 6 class diagram .................................................. 62
3.7 ContactNos 7 class diagram .................................................. 62
3.8 ContactNos 1 metrics results ................................................ 63
3.9 ContactNos 2 versus ContactNos 3 ....................................... 63
3.10 ContactNos 4 versus ContactNos 5 ..................................... 64
3.11 ContactNos 6 versus ContactNos 7 ..................................... 64
4.1 Composition Class Hierarchy ............................................... 71
4.2 Coupling through Methods .................................................. 73
4.3 Coupling through Attributes ............................................... 73
4.4 Class Hierarchy with Extensibility ....................................... 75
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Security Design Metrics Results</td>
<td>65</td>
</tr>
<tr>
<td>4.1 Software Security Metrics Results</td>
<td>85</td>
</tr>
<tr>
<td>5.1 Design Refactoring Rules</td>
<td>93</td>
</tr>
<tr>
<td>5.2 Security-Critical Design Refactoring Rules</td>
<td>94</td>
</tr>
<tr>
<td>5.3 Results for Individual Class Security Metrics</td>
<td>105</td>
</tr>
<tr>
<td>5.4 Results for Multi-Class Security Metrics</td>
<td>105</td>
</tr>
<tr>
<td>5.5 Security-Critical Attributes Design Refactoring Rules Assessment</td>
<td>107</td>
</tr>
<tr>
<td>5.6 Security-Critical Methods Design Refactoring Rules Assessment</td>
<td>108</td>
</tr>
<tr>
<td>5.7 Security-Critical Classes Design Refactoring Rules Assessment</td>
<td>109</td>
</tr>
<tr>
<td>6.1 Program Characteristics</td>
<td>124</td>
</tr>
<tr>
<td>6.2 Data Encapsulation and Cohesion-Based Security Metrics</td>
<td>127</td>
</tr>
<tr>
<td>6.3 Coupling, Composition and Extensibility-Based Security Metrics</td>
<td>127</td>
</tr>
<tr>
<td>6.4 Design Size and Inheritance-Based Security Metrics</td>
<td>129</td>
</tr>
<tr>
<td>7.1 Readability and Writability Security Metrics</td>
<td>134</td>
</tr>
<tr>
<td>7.2 Security Design Principle Metrics</td>
<td>135</td>
</tr>
<tr>
<td>7.3 Total Security Index Metric</td>
<td>136</td>
</tr>
<tr>
<td>7.4 Apache James 2.0.0 to 2.1.3 Data Flow Metrics</td>
<td>140</td>
</tr>
<tr>
<td>7.5 Apache James (1) Readability and Writability Security Metrics</td>
<td>141</td>
</tr>
</tbody>
</table>
Author’s Statement of Originality

This thesis is my original work and has not been submitted, in whole or in part, for a degree at this or any other university. Nor does it contain, to the best of my knowledge and belief, any material published or written by another person, except as acknowledged in the text.

Bandar M. Alshammari
(Author)

December 9, 2011
(Date)
Acknowledgements

First and foremost, I would like to thank Allah almighty for giving me the strength and patience to complete this thesis. I also would like to show my deepest gratitude to my principal supervisor, Professor Colin Fidge, for his continuous and conscious support, guidance and encouragement throughout this project. He has taught me rigorous research, precise writing, effective presentation skills and for the enormous amount of time he spent supervising my PhD studies. My deepest appreciations are also due to my associate supervisor, Dr Diane Corney, for her invaluable feedback and technical suggestions.

I would like to acknowledge the Ministry of Higher Education in Saudi Arabia for sponsoring my PhD studies under King Abdullah Scholarship Program. It is with immense gratitude that I also acknowledge the support and help of Abdulrahman Alsudairy Foundation in Saudi Arabia for giving me the chance and financial support to study overseas after leaving high school.

I must thank as well my colleagues at the Information Security Institute, Bandar Alhaqbani, Aliaad Albeshri, Mohammad Alnatheer, Ali Alhamadan and Mufeed Almarshafi for their advice and friendship.

I am indebted to my late mother, Rafeah Almadhour, for her unconditional love and sacrifice. I would like to thank her for teaching me how to be a caring and loving person. I cannot also forget to express my appreciation to my father, Mzel Alshammari, for his continuous motivation and for believing in me all these years. I must also thank my brothers and sisters for always remembering me.

Last but not least, I cannot find words to express my love and gratitude to my wife, Wafa
Alshammari, for her selfless support and sacrifice she endured during the course of my PhD. I cannot imagine that I could have completed this thesis without her support. I also cannot forget my son, Moayad, who was born while doing my PhD and has inspired me since with his smiles to work hard.
Part I

INTRODUCTION
Introduction

With the increasing demand for developing high-quality and more reliable systems, the process of developing trustworthy computer software is a challenging one. Defining a clear set of requirements that a certain program needs to meet can help in achieving such objectives. In fact, developing and maintaining a system that is guaranteed to meet all essential requirements is a very complicated task.

Computer 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, in addition to maintainability, performance, reusability, and reliability [1]. Many researchers and practitioners admit that although there is no such thing as a completely secure program, there are still ways of reducing security risks and vulnerabilities [2, 3]. It has been identified that security flaws in software are usually due to the circumvention of security mechanisms rather than breaking the security mechanisms themselves [1]. Therefore, it has been suggested that taking security into account from the first stage of the system development life cycle would have a significant impact on decreasing such circumventions [1, 4]. Nevertheless, designers seem to delay consideration of security until a late stage of the system development life cycle [1].

Accepting the fact that no system can be completely secure, in this thesis we address this problem by introducing a set of “security metrics” which allow software designers to measure the system’s information security during the design, coding and refactoring (maintenance) stages of software development.
Chapter 1. Introduction

1.1 Motivation

Systems consist of various assets which require some form of protection or security. Some of these assets need to be protected from being disclosed to unauthorised parties. Other assets are required to be secured from being modified by unauthorised parties. To identify the type of security which is required by a particular asset, studying the system’s architecture is essential. Schneider [4] has defined a number of different design architectures such as software, system, and security. The software architecture is concerned with the computational components and information flow between them [4].

For our purposes, we assume that information flow vulnerabilities occur at the level of the Application Programming Interface (API) and therefore in this research we focus on studying vulnerabilities at the level of program components, often referred to here as ‘programs’ for brevity.

A security-critical program can face a number of threats which could eventually expose its confidential data and hence worsen its security level. Several vulnerabilities can be directly linked to the Application Programming Interface [5]. This means the security of a program can be compromised by an API-level exploit if a sequence of API operations is invoked [6]. These API operations usually rely on low-level details about the execution of the program such as its architecture which can be exploited and cause the program to be vulnerable [6].

The sources of threat this project focuses on are primarily those that exploit vulnerabilities in the software’s Application Programming Interface. These include: malicious programmers who insert security-critical malicious code and exploit it; careless programmers who do not follow specific security coding guidelines and could make security-critical programs easily exploitable; and accidental leakage of data which could be caused by uncontrolled or unintentional events such as the sudden crashing of a security-critical system. Many of these operations can cause a number of unknown vulnerabilities which are due to undesirable flows of classified information, and hence this thesis proposes a number of security metrics that help in eliminating these vulnerabilities before they occur. This project defines a sound approach based on a type system that consists of several typing inference rules to form the foundation and theory of our security metrics.

Our focus is on protecting the confidentiality of data with regard to the software architecture
1.2 Research Problem

(although the principles are also relevant to the closely-related problem of protecting data integrity [7]). Our aim is to quantify security by analysing the flow of information throughout program designs and code with regard to the software security architecture and characteristics, which has some similarities to the work of Rine [8]. There are also other studies which have been conducted to investigate information flow through computer program code. This has been done using several approaches, including type analysis [9] and data/control-flow analysis [10]. However, these studies focus on ‘programming in the small’, i.e., at the level of individual program statements. This project differs from previous work as it focuses on ‘programming in the large’, which takes the overall program module structure into account. This approach allows us to fulfill the desire of easily recognising a secure system by assessing its security level at the design, coding and maintenance stages. To achieve this goal, we introduce a set of security metrics that are capable of easily evaluating a program’s security and can be integrated into a security assessment tool.

The type of system this project concentrates on is object-oriented architectures. Many existing systems nowadays are developed using object-oriented languages such as Java and C#. On the one hand, object-oriented programming languages offer various constructs which help to develop secure software systems [11] such as the feature of data encapsulation [12]. On the other hand, these languages offer other features which could introduce risks to systems as the feature of composition [13].

1.2 Research Problem

By investigating the security properties of individual and composite classes our aim is to define an evaluation technique for easily measuring the relative security of object-oriented programs with respect to their security-critical information flow. Achieving this goal will provide us with a technique suitable for evaluating the security level of a given program at various stages of its development, from design to coding to maintenance.

1.3 Research Questions

The basic research question addressed by this project is as follows.
• How can we measure the relative security of object-oriented programs with respect to their ability to protect security-critical data members?

To solve this problem, a number of decisions needed to be made.

• How do we distinguish security-critical from non-critical program code?

• Are we going to analyse the program’s static or dynamic information flow properties?

• At which abstraction level of the system are we going to assess a given program’s security; statements or classes?

• How are we going to validate the proposed security measurements?

• How can we make the security measurements easy to interpret?

Answers to all of these questions form part of the research presented in this thesis.

1.4 Research Objectives and Approach

The main goal of this project is to define security metrics that can be used to recognise the security characteristics of a given program. Achieving this goal will allow software developers to easily evaluate the security of computer programs and identify sound steps for improving the security of existing programs. The following tasks summarise the main technical objectives of this research project and the approach taken to accomplish them.

1.4.1 Type Inference Model

The starting point for our approach is to identify and describe a type inference model which is the theoretical foundation of both the metrics and the tool. Type inference is frequently used in the information security literature to define data flow through security-critical program code. Such a model allows us to understand how information can flow between classified and unclassified variables, attributes, method parameters and return values in object-oriented programs. This approach also allows us to check the correctness of our security tool which is concerned with tracing information flow within a certain program. Therefore, this type
inference system unambiguously characterises how our tool traces (potential) information flow when calculating security metrics for a given program.

1.4.2 Annotations for Identifying Program’s Security-Critical Features

In practice we begin by identifying the security-critical features of a program or design, as a basis for calculating metrics which evaluate the system’s security level. Such features can be annotated at either the design level or the source code level.

To do this we define some minimal annotations with which programmers can say which parts of their design are meant to be security-critical. From these annotations we calculate our security metrics by combining a static analysis of the design’s structure and an information flow analysis of its security-related variables.

To express security properties for programs based on their design, white box notation is the most appropriate. White box design representations include: structure charts, class and object diagrams, and sequence diagrams [14]. Therefore, we need to use one of the existing software design diagrams to achieve this goal. For our purposes a diagram which shows the general static properties of a class regardless of their interactions, such as a UML class diagram is suitable.

A recent approach developed by Jurjens [15] called UMLsec is a security-related extension of the existing UML notation [16] so we adopt it in our work. It integrates its own features as stereotype and tag value pairs with the existing standard UML diagrams (such as activity diagrams, use case diagrams, and class diagrams) [15]. Its main goal is to assist developers to express system security requirements at the design stage of the system [15]. In our case, we use UMLsec annotations to label attributes which contain confidential data and thus need to be kept secret. We use the UMLsec tag “secrecy” to identify those attributes.

We also need a technique for illustrating how data flows inside and outside a given class based on its design since this information is not usually found in UML diagrams. Such a technique must be able to define the information/data flow between various members (such as methods and variables) inside a given program class. An existing approach is that used by SPARK [17], a high level programming language for reliable and secure programs [17]. SPARK requires the programmer to add annotations to the source code that are helpful for analysing data flow between methods and variables within the same class. For this reason, SPARK’s
annotations seem to be the best solution to our need to add such information to UML designs. Therefore, we adapt these notations to be compatible with object-oriented designs (since the SPARK language is not object-oriented).

1.4.3 Metrics for Quantifying the Security of Object-Oriented Designs

The objective here is to define a technique for evaluating the level of security for a given program based on its design. Since non-trivial programs do not consist of a single class but instead are a composite of several classes, we needed to define metrics that measure compositional properties of security-annotated classes. These compositional properties were based on traditional quality measures such as data encapsulation, cohesion, coupling, composition, extensibility, inheritance and design size. Defining these metrics allows software programmers to assess the security of their programs at an early development stage.

1.4.4 Metrics for Quantifying the Security of Object-Oriented Programs

Another objective of our project is to define a set of metrics which is capable of quantifying the security of a given object-oriented program based on its executable code. Such metrics would make it easy to compare the security of related object-oriented programs. This set of metrics is concerned with assessing the impact of statement level code on the security of a program, by tracing any potential information flow which could occur through interactions between classes and the sequence of method calls.

1.4.5 Transformation Rules for Developing or Maintaining Secure Programs

Since programs are long-lived and undergo frequent maintenance, this objective concerns measuring the impact of software upgrades on system security. Refactoring is a commonly-used approach which introduces modifications in software to enhance its quality without changing its existing behaviour [18] so the focus was on the security implications of applying refactoring rules.

In order to achieve this, we assessed a number of relevant existing refactoring rules to see whether or not they improve or at least maintain the security level of a program and do not introduce new vulnerabilities.
1.5 Research Outcomes

To do this, we first measured the security level of a given program using our security metrics. The program was then refactored according to a set of standard refactoring rules. Finally, we measured the program’s security level again to see how these refactoring rules affected its security. This allowed us to identify which refactoring rules may improve the security of a given program.

1.4.6 Tool Support for Assessing Security-Critical Programs and Designs

The aim here was to provide programmers with a tool which is capable of automatically assessing the security of their programs from the early stages of the program’s development. Such a tool would make it easy to evaluate the security of a given program or design.

1.5 Research Outcomes

This section summarises the main outcomes achieved by this project. To illustrate these outcomes, we use several examples throughout this thesis which have been captured from the designs of real-life applications or existing open-source security-critical programs. We assume that we should be able to detect how real-life programs become more secure as they are debugged. For instance, we use the design of a single class that is responsible for storing private information about a person’s contact numbers. Another example is a simplified design of a defence system which is responsible for storing classified information about a person working within the Department of Defence. Another example is the design of a bank account system class hierarchy that is responsible for storing confidential information about customer accounts. We use these three designs to illustrate our security design metrics and the refactoring rules associated with them. To describe our security code metrics, we analyse a number of open-source security-critical programs to show how they work. All of these programs are security-related, so we could reasonably expect successive releases to be more secure than their predecessors. With regard to describing the refactoring rules associated with the code-level, we use a program inspired by existing refactoring examples in the literature. We use our security metrics to study the impact of a number of standard refactoring rules on security by evaluating the metrics before and after refactoring. This approach allows us to identify which strategies/rules need to be taken in order to improve the overall security of a given program.
by treating the refactoring rules that improve our security metrics as general strategies for mitigating API-level security vulnerabilities.

1.5.1 Type System

One of the main outcomes of this project is a security type system consisting of a number of type inference rules which cover all of the relevant constructs for most object-oriented programming languages as shown in Chapter 9. These rules describe how security levels are assigned to variables and statements in a given program. This type system formally defines the way in which our analysis tool traces information flow in order to measure the relative security of object-oriented programs.

1.5.2 Security Annotations

A set of minimal security annotations is one outcome of this project as shown in Chapters 3, 4 and 6. They are used by the programmers to identify which data needs to be kept confidential in the design and coding stages of the program’s development. They are also used to express the interactions between attributes and methods for a program’s design. These annotations express the programmer’s security intentions and form the starting point for calculating our security metrics.

1.5.3 Security Measurements

Two sets of metrics have been defined by this project to quantify the overall security of object-oriented programs. One set is capable of assessing the security of a program based on its UML class diagrams as defined in Chapters 3 and 4. The other set evaluates the level of security for a given source code program as defined in Chapters 6 and 7. Both sets of metrics can be used to compare various versions of the same program (or design) and show which version is more secure.
1.5.4 Secure Refactoring Rules

The metrics were used to assess the security impact of a number of standard refactoring rules on existing computer programs. This assessment allows us to identify a set of rules that can be used to change the existing structure of a computer program while still preserving or even improving its current security level as shown in Chapters 5 and 8. (Additionally, this set of refactoring rules can provide guidance for developing secure programs.)

1.5.5 A Security Assessment Tool

A software tool was implemented to provide software developers with an easy aid for designing security-critical programs, by giving them the ability to assess their designs with regard to our security metrics automatically shown in Chapter 10. It can statically analyse annotated UML designs and Java programs in order to extract our security code metrics and thereby assess the program’s security level.

1.6 Thesis Outline

This thesis is divided into five parts which describe the main outcomes of this project as per Figure 1.1. Part one consists of two chapters which briefly introduce the main motivation and objectives of this project, and describe recent work on software security, respectively. Chapter 2 also touches on various approaches to producing more secure systems. This includes established general principles for designing secure systems. It also shows how refactoring is generally used for enhancing the quality of existing programs including security. Lastly, it provides an introduction to general software quality measurements for maintainability and usability including previous software security metrics. This part also discusses different methods for validating software metrics and the challenges facing these methods.

Part Two consists of three chapters (Chapters 3, 4 and 5), which describe our approach for assessing the security of object-oriented designs. Chapter 3 focuses on the security design of individual classes for object-oriented applications. It defines seven security metrics that measure Data Encapsulation (accessibility) and Cohesion (interactions) of a given object-oriented class from the point of view of potential information flow from security-critical fields. Designs are
annotated using UMLsec and SPARK’s annotations to identify which data needs to be kept secret and to express the information flow relations between attributes, methods, and classes of a given design. Chapter 4 extends these metrics to define another set of security metrics for multi-class programs. These metrics are defined built on five properties which are related to the overall design of an object-oriented program: composition, coupling, extensibility, inheritance, and design size. Chapter 5 uses these two types of metrics to assess the impact of a number of standard refactoring rules on the security of refactored designs of a given program by evaluating the metrics before and after refactoring. This allows us to identify which refactoring rules can improve the security level of a given program from the point of view of potential information flow.

Part Three also has three chapters (Chapters 6 to 8), which define an approach for measuring the security of object-oriented programs based on the analysis of their source code. Java is chosen as the illustrative object-oriented programming language. Chapter 6 defines a set of object-oriented Java security metrics which are capable of assessing the security of a program from the point of view of potential information flow. Chapter 7 presents a hierarchical model
for assessing an object-oriented program’s security at different levels of abstraction based on a static analysis of its code. The model consists of four hierarchical levels, and is validated via an experiment involving a number of open source Java programs. Chapter 8 assesses the impact of common code level refactoring rules on security, allowing us to identify which refactoring rules can improve the security of a given source code program.

Part Four then introduces automated support for automatically measuring the security of object-oriented programs based on their designs or source code. Chapter 9 describes a type-inference system for defining precisely what we mean by secure data flow. This chapter defines the theoretical foundation of both our security metrics and security assessment tool. It formally defines the capabilities of our tool via a number of type inference rules which cover all of the relevant Java language constructs. Chapter 10 describes our tool’s implementation and how it analyses Java programs’ compiled bytecode. It also shows how our tool converts security-critical UML designs to equivalent Java source code and then bytecode so the tool can measure the security of a program at all three of these levels.

Finally, Part Five concludes this thesis. It consists of Chapter 11 which summarises the main contributions of this project as described in Chapters 3 to 10. This chapter also provides directions for possible future work related to this area of research.
Chapter 2

Literature Review

In this chapter we review previous research relevant to our goal of developing security metrics for object-oriented programs. In particular, we review established security design principles and previous metrics that formed the starting point for our research.

2.1 Introduction

Much existing software is designed with poor consideration of information security which makes it vulnerable to many threats including malicious attacks [19]. Software patches are one of the suggested solutions for many of the security attacks facing software [19] but they are expensive to develop and deploy and do not solve basic design weaknesses in the program code. Another solution to achieve a secure product is by following a trustworthy security process [3]. Security processes, in general, consider many aspects of system design, coding, testing, and auditing [3] (e.g., international security standards such as the Common Criteria [20] or the Trusted Computer Criteria [21]). Another common approach for achieving a secure computer program is by following certain coding guidelines which focus on the level of individual program statements (e.g., to avoid/detect buffer overflows [22]). However, these solutions do not always work effectively and may, in general, even introduce new vulnerabilities to existing software [19]. Adding security features to systems after they have been developed and deployed has been a major cause of many system vulnerabilities [23]. Therefore, applying security principles from the early stages of the software development life cycle (SDLC) would be a better solution [19] and allow a more coherent system to be produced [23].
Developing a secure system requires a good overall design which takes security into account from the beginning. Security-critical design decisions aim to provide guidance to software designers to ensure that system security is being carried out the right way [12]. Gollmann proposes a series of such design decisions as follows [12].

- Decision one states that it is important to clarify what needs to be protected, whether it be data, operation, or users. In practice, most operating systems are concerned with protecting data while commercial applications are mostly interested in controlling users’ activities [12].

- Decision two involves deciding on which computer system layer the protection mechanism is placed. Computer systems typically consist of five different layers: Hardware, OS Kernel, Operating System, Services, and Application [12]. However, to make such a decision, it is necessary to carefully look at the type of the system, its purpose for existing, and its most important concerns.

- Decision three aims to identify the main security functionality preference, whether it is simplicity and higher assurance or a feature-rich security environment. To reach a high level of assurance, a secure system needs to be shown to adhere to several system design principles. The simpler the system is, the higher assurance level it can reach [12].

- Decision four is to identify who should be responsible for handling each security mechanism; should control be handed over to a central entity or to the individual components. Both of those solutions have advantages and disadvantages. A central entity may be a bad choice if performance is a main concern although it is a good solution when it comes to adhering to a consistent policy [12]. On the other hand, a distributed solution tends to be a good choice in terms of performance but makes it hard to prove that the system adheres to a consistent security policy [12].

- Last but not least is deciding how the attacker can be prevented from having access to a layer below the protection mechanism. Access to such a layer can provide control over that layer which could lead to the protection mechanism being compromised [12].

A suggested methodology by Fernandez [19] incorporates security principles into each stage of the SDLC. It makes sure that each stage complies with these principles through testing
cases. Another methodology by Eduardo and Xiaohong [24] called ‘security patterns’ is used to improve the security of software based on existing knowledge of software attacks. These patterns aim to enforce security at the application level of a given program [24]. Preventing unauthorised access to sensitive data is the main goal of this approach [24]. Other security patterns have also been defined elsewhere including Schumacher, Fernandez-Buglioni, and Hybertson’s book [25] which shows how security patterns can be a useful solution for many recurring security problems when designing systems. It includes a survey of existing security patterns for wireless networks, agent systems, and enterprise systems. The authors claim that patterns, in particular, are not only restricted to be used by systems’ designers but also systems’ architects, developers, and implementers [25]. One of these patterns is the standard security concept of Mandatory Access Control (MAC) which has been modelled using UML class diagrams [25]. Thuong et al [26] have shown a mathematical approach of incorporating MAC into three types of UML diagrams (class, sequence, and use case).

Nevertheless, the main issue is how to express these patterns and principles in a language or notation which is easy to understand and implement. Jurjens [27, 15, 28, 29] proposed an approach called UMLsec which relies on integration of multilevel secure systems ideas and security protocols standards to the Unified Modelling language (UML). UMLsec aims to highlight the possible vulnerabilities of a given system. Therefore, it helps document the security-critical features of the system during the development process. This approach uses four different types of UML diagrams [27]. Class diagrams are used to make sure that security level rules are obeyed when exchanging data within the system. State chart diagrams are used to show a dynamic view of objects’ behaviour which helps in making sure that no high level data is indirectly sent to low level objects. Interaction diagrams show interactions between objects including security-critical interaction. All of these requirements need to be implemented at the physical level which is the responsibility of deployment diagrams. In addition, indications of security levels within the system are done by using tags [27, 15]. Finally a verification framework has been automated for verifying UMLsec models for security requirements. [28, 29].

Another type of notation defined by Peterson et al. [23] is called UMLpac and aims to integrate security features into a system’s class design. It creates a ‘security package’ which covers the system’s security features to separate the system’s class diagram from the system security package, thus keeping a level of abstraction between them [23]. This approach
introduces a catalogue which contains the identified system threats, their risk factors, how they can be prevented, and help developers avoid certain insecure coding practices [23].

Although these notations can be a starting point for developing secure software, since they can identify data which needs to be kept confidential, they do not provide an approach for quantifying the security of a given program or design. We therefore need security metrics which respect established security design principles to provide guidance for designing and recognising secure systems. Such metrics will also allow us to identify which refactoring rules are likely to enhance the security of a given program. In the following sections, we will review security design principles, software quality metrics and software refactoring principles relevant to achieving this goal in more detail.

2.2 Software Security Design Principles

To deliver a good software design, software design principles should be considered before the system is developed. They aim to provide guidance for software engineers to increase the assurance of software quality and therefore increase the software’s security [30, 31]. Furthermore, the mechanisms that make a program secure should be self evident in its design. Bishop’s [32] and McGraw’s [11] texts are two recent examples of books that have identified several general principles which help to produce secure software. However, the work of Saltzer and Schroeder [33] in 1975 was one of the first on software security design principles. In this project, our concern is to identify secure design principles at the object level. We aim to develop a security evaluation technique that will help us in assessing the security of a given program and introducing good security design practices into an existing program.

Security design principles can be described as concepts or guidance which can be followed to develop more secure systems at the software design stage. A fundamental design principle is that systems should be as simple as possible. Simplicity makes it easy to understand the system design and security mechanisms and decreases the chances for inconsistencies with the defined system policies [32]. It also reduces the interactions of system components which results in fewer security checks [32]. Security design principles must also enforce some form of restrictions, to reduce the power of a subject to get what is needed only and to communicate only when it is necessary [32].
Generally, a design principle’s main goal is to increase our assurance in the quality of software, which in turn can increase the security of certain software [30]. It is widely known that well structured programs are most likely to be more comprehensible and easy to verify correct and secure than poorly structured programs [30].

However, since we aim to measure security statically, based on the program’s design, rather than its dynamic behaviour, we need to consider which of these established principles are applicable to the program’s static structure and which ones are applicable to its dynamic behaviour only. Below we review a wide range of secure software design principles and then summarise those that are relevant to our research.

2.2.1 Principle 1: Architecture Matches Program

Architecture is concerned with architectural elements selection, how these elements interact, and what their constraints are [34]. These are to be constructed in a way that ensures satisfaction of the system’s requirements [34]. To enhance system verifiability, the architecture should be structured in a way that matches the programs that are built upon it [30], making it easier to check system correctness, which would lead to an increase in the system’s security level. This approach has been recommended to be used within hardware design [30], but can also be applied to software design. Schaitano et al. [1] stated that security engineers are responsible for defining secure systems architectures, which are capable of preventing or at least controlling any malicious attacks. They have developed mail server software that has considered security from the beginning of its development life cycle [1]. The case study has shown how the software prevented many attacks which other systems failed to prevent. Furthermore, a framework introduced by Daniel Conte de, et al. [35] proposed to construct secure systems at the architectural level. It aims to avoid security vulnerabilities within the system at the architecture level [35].

2.2.2 Principle 2: Secure the Weakest Link

It is obvious that hackers will look for parts of the system which seem to be weak and are easy to break [11]. Therefore, this principle says that the weakest parts of the system are the ones which should be secured intensively. It is known that the system’s weakest parts are often those which rely on human intervention by, for instance, administrators, users, and technical support
staff [11]. From real examples a small mail server would be more likely to be a favourable target to many hackers to hack than a bank mail server (but also of less value).

One way to secure those parts is to consider applying cryptography [11]. This would not make those parts 100% safe, but cryptography can be structured so as to require huge computational effort and knowledge to defeat [11]. Cryptography aims to minimise the ‘security perimeter’ [30], which minimises what needs to be verified secure [30]. This leads to fewer security-critical functions which means less exposure to threat. Memory corruption vulnerabilities can be described as a weak link of many software systems which, according to this principle, needs some extra protection. Buffer overflow is the most common such vulnerability and is considered to be “the single biggest software security threat” [11, 36].

Buffer overflow occurs when data is copied to a place in memory that is bigger than the reserved size [36]. Its threat arises from the fact that such a bug can have an effect on the system’s data integrity [36]. Common examples of buffer overflows are stack overflows and heap overflows. It has been described as a design problem which occurs mostly when certain codes or functions are used [11]. It is more likely to be seen in languages such as C and C++ but is hardly seen in modern languages such as Java which provide bounds checks on arrays and pointer references [11].

Avoiding buffer overflows seems to be easy in theory but has proven hard in practice. One of the best ways to avoid such a threat is to avoid certain vulnerable functions and replace them with others that do the same job. In “Building Secure Software” Viega and McGraw [11] list most of the functions found to cause buffer overflows and the alternative functions which avoid it. Functions such as gets, strcpy, and strcat, can be replaced with fgets(buf, size, stdin), strncpy, and strncat respectively [11].

2.2.3 Principle 3: Least Privilege

Bishop defines this principle as “a subject should be given only those privileges that it needs in order to complete its task” [32]. This principle not only relates to computer security but also to other security disciplines such as military security. An important principle in the military is declared as “the need to know” [33]. Bishop adds to this principle by stating that a subject should have only append rights when it needs to only append to the information already contained in an object [32].
This principle can be described as “programs and users should run with the least privilege to complete their job” [33]. The main advantage of adhering to this is that it can limit damage from implementation errors or attacks [33]. Another advantage is to minimize the interactions among privileged programs [33]. It is claimed that designing protection systems this way will help to identify where privileges and transitions should go and also where firewalls should be placed [33].

In real world systems, this principle rarely covers all the different aspects and features of the system. It is usually applied partially which means the risk of being exposed to threats still exists [32]. An example can be seen in the Windows and UNIX operating systems which do not apply access controls to the root user who is able to create, delete, read, and write any files. Consequently, a higher risk of compromising data and records exists [32].

Another principle similar to this is one defined as the principle of least authority (POLA). It suggests that objects having access to a certain component might not have that right for other components [37]. A capability based protection system is used to accomplish this by reducing the authority of objects [37].

2.2.4 Principle 4: Fail Safe Defaults

Bishop defines this principle as “unless a subject is given explicit access to an object, it should be denied access to that object” [32]. Saltzer stated [33] that this principle was suggested by E. Glaser in 1965 so that access decisions are based on permission rather than on exclusion [33]. It also means that the default situation is lack of access and the protection mechanism scheme should identify conditions under which access is permitted [33]. Moreover, the reverse, in which the default situation is full access, is risky since it would be much harder to find errors and mistakes with a system that does not exclude users’ rights [33]. However, designers using this principle should always follow a conservative design which is based on arguments for why objects should be accessible rather than why they should not be [33].

Moreover, Bishop adds that to be in a more secure environment, the subject should undo any changes it has made in the system security state before it terminates when it fails to complete its task [32]. For instance, the protection of a spool directory should be classified in two parts; one is to give create and write privileges to the mail server, and the other is to give only read and delete privilege to the local server [32]. However, if a mail server has failed to create a
Chapter 2. Literature Review

new folder in the spool directory then it should roll up its activities, and should not be allowed
to extend its rights or place the new folder somewhere else in the directory. If such an action
exists in the system then a hacker could take advantage of it either by overwriting other records
or filling up other disks [32].

2.2.5 Principle 5: Complete Mediation

This principle states that every access to any object must be checked and identified [32, 33].
It assumes that authority may change at any time and so may access levels in a security
mechanism. As a result of that, it is vital to construct a solid plan to check identity and authority
at every request [33]. A system that is designed to check identity for every request can decrease
the chances of mistakenly providing permissions to unauthorized subjects [38]. It also makes
the system unfavourable to many attackers to hack [38]. By this principle, caching information,
which is a method widely used to speed up the identity process after the first access, should be
considered very carefully [32, 33].

Bishop mentions an example when no second check on access permission is performed and
cached values are used instead [32]. This is when a UNIX process tries to read a file which has
the permission to read. After making that request, the process will receive a descriptor encoding
the allowed access. The process has to show this descriptor to the kernel for every access. The
issue is raised when the owner of the file disallows the access of that process. If the kernel is
using caching then it will get the cached value and allow access. This happens when there is no
second check performed thus violating this principle.

Single Sign On is an approach which ensures that passwords have been only entered
once. Thus, the system will be responsible for authenticating the user whenever needed [12].
Although this approach seems to be attractive for users, it also raises security concerns. As
a result of that, convenient design is not always favourable especially when it conflicts with
security design [12].

2.2.6 Principle 6: Economy of Mechanism

Bishop defines this principle as “security mechanisms should be as simple as possible” [32].
This principle is important during the software design process due to the fact that unnecessary
information or control flow paths could result from an overly complex design [33]. To make the design simple, known components of good quality should be reused in the system whenever possible [11]. To solve the problem of existing components of unknown quality, the system should be inspected carefully and there should be some sort of physical examination of hardware parts which implement the protection mechanism [33].

The refactoring approach can be a solution to make the problem of making a design and implementation easier to prove safe. 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 behaviour” [18]. To improve simplicity while ensuring security, the system should use a small number of “Choke Points” [11]. Those are used to force the users to follow a certain path along which security is improved [11].

2.2.7 Principle 7: Open Design

There is no protection mechanism that can be hidden forever [33]. Consequently, this principle states that the security of a mechanism should not depend on how the system design and implementation have been hidden [32]. This is highly related to software which is widely distributed, such as commercial software [33]. It has been suggested that a better solution than hiding secrets is to use more easily protected keys and passwords [33]. A public protection mechanism allows it to be examined and reviewed by many other experts without the fear of the system being compromised [33]. In addition, interested professional users can examine the system for themselves to prove that they are using what they are aiming for [33]. Moreover, keeping the cryptographic passwords and keys secret does not conflict with this principle since they are not themselves algorithms [32].

(A counter approach to this principle is “security through obscurity” where the believe is that code secrecy can enhance system security [32, 39].)

2.2.8 Principle 8: Separation of Privileges

This principle states that the system should grant permissions to access resources only after two conditions or more are met [32, 33]. This principle was pointed out in 1973 by R. Needham. It claims that no single condition or error should be capable of breaching system security [33]. For
instance, once the system is locked then the system keys should be separated in two different directions, for example as in bank deposit boxes. In computer systems, a net banking system needs two questions answered in order to perform a bank transfer. Unfortunately, such an approach is not always favourable and convenient for customers [33].

2.2.9 Principle 9: Least Common Mechanism

This principle is defined as a “mechanism used to access resources should not be shared” [32]. The principle claims that shared mechanisms provide an unnecessary information on control flow path between users. Such paths should be minimised to better secure the system [32]. However, if there is a mechanism that has to be shared between users then it has to satisfy the concerns of all of its users [33], for instance security concerns. It is also stated that shared functions should be run in users’ interfaces rather than as a system procedure [33].

An example of this principle is sharing a website which offers electronic commerce services with all of the users. An attacker having access to such a website will flood it with messages till the website goes down. Customers will not be able to access the website at this stage which is a loss to the business [32].

2.2.10 Principle 10: Psychological Acceptability

Bishop defines the principle of psychological acceptability as a “security mechanism should not make the resource more difficult to access than if the security mechanism were not present” [32]. This has been justified by observing that simple resources will be used as intended while complicated resources could lead to users to compromise the system unintentionally [32, 33]. Error messages should state the exact problem without any additional unnecessary information. However, giving some unnecessary information could lead to users misusing the system [32]. Poorly designed error messages could state where in the program code the problem occurred instead of giving information about what the problem is and how to solve it. In summary, this principle does not ignore the fact that a security mechanism may add some extra complexity to the system but this is acceptable as long as it is logical and is reduced to the minimum possible [32].
2.2.11 Principle 11: Work Factor

This is a physical security principle which, however, seems to be of relatively little applicability when applied to computer systems [33]. The work factor principle, as Saltzer states, represents the cost of circumventing a defence [33]. It is based on comparing the cost of circumventing (work factor) with the resources of a potential attacker [33]. It should cost more to compromise a system than it is really worth.

A relevant example would be when a user needs to enter a password consisting of four case-sensitive letters. This would result in $52^4 = 7,311,616$ possible combinations for that password. If it is assumed that the attacker enters a password manually every 5 seconds, this would take more than a year of continuous work. Such a combination seems like a reasonable safeguard. Nevertheless, this is not the case if the attacker is using software which generates millions of passwords every second.

In real world practice, there are always different ways to make it cost less to compromise a system. From the previous example, if the attacker is using automation to generate passwords then it would cost less in terms of time and resources usage. However, the work factor is not easy to calculate in many cases since there are no defined standards which can be followed [33]. Another disadvantage of this principle is that it only takes into account the work factor resulting from systematic attacks, which many computer protection mechanisms are qualified enough to defeat [33]. However, it is not easy to calculate the work factor based on attacks resulting from either waiting for a system failure or an implementation error since predicting when such events will occur is impossible [33].

2.2.12 Principle 12: Compromise Record

This is another design principle which Saltzer [33] suggested cannot be used in computer protection mechanisms. This principle suggests that if a security plan or mechanism has been compromised then it is necessary to record all the breaches and changes that occurred during the hacking process [33]. Such a strategy would aim to launch another security plan that can prevent the previous breaches, thus making the system more secure than before. An example of such case is when a particular folder has been modified by an unauthorized user. The owner will discover such a violation sooner or later and will then secure the folder against future misuse.
This particular approach is rarely used in computer protection mechanisms due to the fact that it is hard to find out that the system has been broken once the incident occurred \[33\] and it is difficult to automate a response.

### 2.2.13 Principle 13: Practice Defence in Depth

This principle is based on the concept of Defence In-Depth which consists of layering protections so that the compromise of one aspect of a system is mitigated by other controls \[36\]. The main goal of this principle is to manage security risks by a variety of effective tactics \[11\]. As a result of that, redundancy of security layers is a strategy used by this principle to acquire a higher level of security \[11\]. An example of this concept is using low privileged accounts to run services and domains, and isolating different functions to different pieces of hardware \[36\].

Viega and McGraw mention a real situation which can use this principle to produce a safer system. This is to place a firewall around a corporate application server in addition to the corporate-wide firewall. This would guarantee a higher security level whenever the server is communicating with the corporate database \[11\]. To increase the level of security even further during this process, the data in the database can be encrypted which means another layer of protection \[11\].

### 2.2.14 Principle 14: Fail Securely

Systems failures are not avoidable but security issues related to that failure can be avoided \[11\]. Some risks only occur when the system fails whereas if a system is normally working then no problems should be faced \[11\]. However, it is necessary to ensure that if a certain part of a system fails then it should fail securely \[11\].

Barnum and Gegick have identified several strategies a system should follow after it fails to ensure a secure failure \[38\]:

- Ensure secure defaults (i.e., deny access).
- Undo all changes and restore to a secure state.
- Always make sure to check values for failure.
2.2. Software Security Design Principles

Figure 2.1: Fail Securely Example

- A default case which performs the right thing in a conditional statement.

It is necessary to verify how the system behaves after it fails and make sure that the failure does not harm the system [38]. A possible case is revealing sensitive information after a system fails which helps attackers to establish an attack [38]. Howard and LeBlanc verify that revealing unnecessary information about a system failure could help attackers [3]. The golden rule when failing securely is “to deny by default and allow only once you have verified the conditions to allow” [3]. The following example explains [38] the golden rule stated by Howard and LeBlanc [3].

The code in Figure 2.1 shows a conditional statement that gives access to a user depending on the return value of variable dwRet. The code in the bottom half of Figure 2.1 can work perfectly but a problem will occur if method IsAccessAllowed, which decides whether to give access or not, fails for a reason such as “ERROR NOT ENOUGH MEMORY” [38]. In this case, the user will grant access because result dwRet is not equal to “ERROR ACCESS DENIED” [38]. On the other hand, the first code fragment in Figure 2.1 shows that if method IsAccessAllowed fails for any reason, no access would be granted [38].

2.2.15 Principle 15: Compartmentalize/Isolation

This principle’s main goal is minimising the amount of damage to a system [11]. To achieve this, it considers two aspects. One is breaking up the system into small sub systems [11]. The
second is to isolate program code which has security privileges [11]. This principle, in general, is better than systems that use access control mechanisms that allow all types of access or none [11].

UNIX’s privilege model represents a bad example of compartmentalization since it gives users an all or nothing access model [11]. If a UNIX user has root privilege then this user can do anything to the system even if this capability is not needed for the particular job at hand [11]. An example of this privilege model is that it is not possible to be bonded to port 1024 on UNIX systems unless the user has root privilege [11].

Another similar principle is isolation. Isolation has been found useful in hardware security designs since its main goal is to enforce a partial isolation of domains [30]. This means that those isolated domains will interact only with those domains and environments allowed by the security policy [30]. To achieve such isolation, there are four suggested ways: temporally, physically, cryptographically, or logically [30].

2.2.16 Principle 16: Reduce the Size of the Attack Surface

Howard [40] has identified several techniques to reduce the size of the attack surface of a given system. One is to reduce the amount of running code. This can be done by turning off some unnecessary features by deploying the 80/20 rule, which consists of eliminating those features not used by 80 percent of the system’s users. Another of these techniques is to reduce access to entry points by untrusted users, which can be achieved by authenticating all users of certain entry points. Another technique is to reduce privileges to limit potential damage, i.e., to reduce the privileges under which certain functionalities and processes execute. Another technique aims to identify threat code/design paths. This is part of the threat modelling process which can be accomplished through UML or DFD diagrams. Tracing these paths is capable of identifying the data an attacker can access. The last technique is to measure the attack surface. This should be done each time a system has faced a change to either its environment or its functionalities. If there is an increase in the size of the attack surface, then it is better to identify the reasons behind it and try to reduce the size. The best approach is to define the minimal attack surface early in development, and then measure it regularly during the system development life cycle. Howard [40] has also stated two approaches to measure the attack surface size. One is counting the number of resources that contribute to the attack surface such as functionalities and system
channels. However, this is a misleading approach since it assumes that all the resources make the same contribution to the attack surface. The second is using a predefined metric that takes into account the resources contributing to the attack surface. Also, the metric should assess each resource damage potential-effort ratio, as in the approach of Manadhata et al. [41, 42, 43]. This approach calculates the sum of three resources and their damage potential-effort ratio: (1) the system’s entry and exit point, (2) the system’s channels (those used to connect to the system), and (3) the system’s untrusted data items (those items which an entry or exit point has direct access to) [40].

For our purposes, we define the attack surface to comprise the program segment’s Application Programming Interface as well as its inputs/outputs. This means that a program should have the least relative proportion of attackable code which consists of readable security-critical attributes, methods and classes and methods which can write security-critical data externally.

### 2.2.17 Summary of Relevant Design Principles

Below we identify which of these security design principles we think are measurable by static analysis, including information flow analysis. The remaining ones are those which we think are either not measurable or only measurable by dynamic analysis of a given program.

**Grant least privilege** To adhere to this principle, systems must restrict the privileges of their users to the least possible [32]. For object-oriented programs, this means a program that has the necessary functionality but whose methods can do the fewest possible actions is the most secure. In our case, a program whose security-critical data is writable from the fewest attributes, and by the fewest methods and classes is considered to be a secure program with respect to this principle.

**Reduce the Attack Surface** The principle of reducing the size of the attack surface aims to limit access to secret data [3]. For our purposes, this means a program should have the fewest possible readable security-critical attributes, methods and classes. It should also mean that there are as few methods as possible that can write security-critical data externally.
Secure the Weakest Link  As a result of having less functionality, a program would have less security exposure, and hence would satisfy the principle of securing the weakest link [11]. In our case, a program with the fewest writable security-critical attributes and methods would satisfy the requirement of this principle.

Fail-Safe Defaults  This principle aims to reduce the capabilities of individual classes with regard to the readability of security-critical data [32]. Hence, for a program to adhere to this principle, each class should have the fewest classified attributes and methods which can access classified data.

Least Common Mechanism  This principle indicates that security-critical data can be transmitted to unauthorised parties through shared resources, and so such sharing should be minimised [32]. In our case, this means that an object-oriented program with the least readability and writability of security-critical classes would adhere to the requirement of this principle.

Isolation  The principle of isolation’s main goal is to minimise the amount of potential damage to a system via enforcement of security privileges [11]. To achieve this goal, we have to consider two aspects. One is breaking up the system into small subsystems [3]. The second is to isolate code which has security privileges [3]. In our case, an object-oriented program which has the least writability of security-critical classes would adhere to the security principle of isolation.

Economy of Mechanism  This principle is important during the software design process due to the fact that unintended information flow paths could result from a complex design [33]. This complexity could result in design and implementation errors which make the entire system or part of it more vulnerable. For our purposes, a program with the fewest numbers of security-critical attributes, methods, and classes would satisfy the economy of mechanism principle.

2.3 Software Quality Measurements

Several studies have developed metrics for quantifying software quality attributes of object-oriented applications such as reusability and functionality [44, 45, 46, 47, 48, 49]. These metrics
aim to measure a certain quality attribute or a set of attributes at various stages of a program’s
development life cycle.

An early study conducted in 1989 by Morris [50] suggested a number of object-oriented
metrics. This was followed by Chidamber and Kemerer’s work [44, 45] which identified a
metrics suite for object-oriented designs, including metrics for weighted methods per class,
coupling, cohesion, and others. The influence of these metrics on finding software weaknesses
at the design stage of a program was analysed by Subramanyam and Krishnan [51]. A
recent work has validated the metrics suite developed by Chidamber and Kemerer on six Java
open source programs for a number of object-oriented quality attributes including reusability,
derstandability, testability and maintainability [52].

Briand et al. [53] suggested a modification to the cohesion metric identified by Chidamber
et al. which considered attributes that are not accessed by any other methods. In a later
work, Briand et al. [54] defined a unified framework for measuring coupling in object-oriented
programs.

Bansiya [55] also identified another approach to measure software cohesion and complexity
at the design stage of a program by analysing class’s method’s signatures. Bansiya and Davis
later suggested an approach to improve Dormey’s Quality Model for Object-Oriented Design
(QMOOD) [56]. This approach aims to measure the quality of various object-oriented design
attributes such as reusability, flexibility, and functionality. Even though this approach covers
most design quality attributes, it does not consider security as one of these attributes.

2.3.1 Software Security Metrics

Many software quality attributes have been studied and measured, including maintainability,
performance, reusability, and reliability [56]. Security, on the other hand, has received relatively
little attention. A common approach which is used by many programmers to assess the security
level of a given program is based on the identification of pre-existing vulnerabilities [57, 58
3 59 60 61 31]. The National Vulnerability Database of the National Institute of Standards
and Technology [62] classifies software vulnerabilities into eight different classes based on the
cause of the vulnerability, namely Input Validation Error, Access Validation Error, Exceptional
Condition Error Handling, Environmental Error, Race Condition Error, Configuration Error,
Design Error and other errors which don’t belong to any of the above classification.
Another technique used by Maruyama [58] and Howard and LeBlanc [3] aims to assess the level of security of given program code. This technique classifies code as either secure or not secure. Secure codes are those that do not introduce vulnerabilities to the system and insecure codes are the opposite. However, this technique does not distinguish between programs that are partially secure or partially not. In addition, it does not quantify how secure the code is. Therefore, there is still a need to establish a metric-based software security model [2] to assess the level of security for a given program or object-oriented class. Furthermore, most previous security measurements which have been defined either assess security at the abstract system architecture level [41] or at the low level of individual code structures [63]. The following sections briefly describe previous work on software security metrics at various stages of the software development life cycle.

**Architecture Level Security Metrics** Measuring the security of the system’s architecture is an important aspect of identifying the overall security of a given program. One of the studies in this field is by Antonino et al. [64] who define an evaluation technique for measuring the security of an existing service-oriented architecture. This evaluation technique is based on two types of metrics: severity and credibility. Severity relates to the value of tagged security artifacts while credibility is the probability of correctly assigning a tag to its relevant system component [64].

Further work in this area was conducted by Liu et al. who proposed a model called the “User System Interaction Effect (USIE)” [65]. The USIE model is responsible for providing a systematic approach to identify security defects from the architecture of a service-oriented system [65].

A recent approach for measuring security based on an architecture by Manadhata et al. [41] measures security with regard to the attack surface size, as described above. The system’s attack surface measurement is an indicator of the risk of attack [43]. It is based on the set of possible resources which an attacker could use to attack the system [43], including methods, data and channels [43]. A method is described by Manadhata et al. as a system entity which could send data (exit point) or receive data (entry point) [41]. Data in their approach is any entity which is visible in the current system such as files, cookies and database records [41]. They also define channels as system entities which can be used by an attacker to invoke the system’s methods such as sockets and pipes [43]. A smaller attack surface indicates a smaller number of potential attacks, and thus a more secure system. They use this metric to compare the attack surface size
2.3. Software Quality Measurements

of different versions of two IMAP servers and two open source FTP demons [43].

Design Level Security Metrics  Measuring security at the design phase, based on typical
design artifacts, has not been considered until recently even though such metrics could
efficiently eliminate software security vulnerabilities before they reach the final product [1 4].
Such metrics would also allow software developers to compare the security level of various
alternative designs under consideration.

A recent proposed framework by Chandra and Khan [66] aims to provide software
developers with systematic guidance for developing and validating security design metrics.
The framework is classified into a number of factors including the identification of security
requirements, vulnerabilities, metrics, and the validation model [66].

Another work in this field was by Agrawal et al. [67] who defined a measurement of object-
oriented class vulnerabilities. The measurement aims to count the number of vulnerable classes
in a given design [67]. A vulnerable class is the one which has sensitive and confidential
data members and methods [67]. In other work Argawal and Khan studied how inheritance
can worsen the security of a given object-oriented design by extending vulnerable attributes
to other classes [68]. They defined vulnerable attributes as those which provide entry points
to confidential data, and their proportion in a design is calculated by dividing the number of
vulnerable classes to the total number of classes in a hierarchy [68].

Code Level Security Metrics  Developing security metrics at the level of source code is
another common approach for quantifying security of a given program. Chowdhury et al. [63]
defined a number of security metrics that assess the security of a given program based on code
inspections. These metrics consist of Stall Ratio, Coupling Corruption Propagation and Critical
Element Ratio. They define Stall Ratio as the number of lines of non progressive statements in
a loop to the total number of lines in that loop [63]. Coupling Corruption Propagation measures
the coupling between methods and their parameters, and is defined as the number of child
methods which are invoked with their parent’s method parameters [63]. The Critical Element
Ratio is the ratio of the critical data elements in a given object to the total number of elements in
that object [63]. They demonstrated the applicability of these metrics on two different Eclipse
plug-ins, Java Pathfinder (JPF) and JDemo Launch.
Another similar work has been conducted by Aggarwal et al. [69], who indicated that unhandled exceptions could cause potential vulnerabilities and thus a less secure program. Their metric is a ratio of the number of handled catch statements to the total number of possible catch statements in a given program [69].

Alves-Foss and Barbosa [70] proposed another code level security metric called the Software Vulnerability Index (SVI). The metric depends on evaluating a number of factors such as system characteristics, potentially neglectful acts and potentially malevolent acts [70]. The metrics are a ratio between zero and one. Higher values of the SVI indicate a higher vulnerability level, and hence a less secure system [70].

A similar method to Alves-Foss and Barbosa’s [70] is Alhazemi et al.’s [71] code level security metric called Vulnerability Density (VD). This metric aims to predict the number of potential software vulnerabilities in a program by inspecting its code, and is calculated as the ratio of the number of vulnerabilities in a given program to the size of the program [71]. The authors validate their metric on five operating systems consisting of three successive versions of Windows and two versions of Red Hat Linux [71].

2.3.2 Validation of Software Metrics

There exist a number of approaches which have been used to validate software metrics. One of these is a methodology proposed by Schneidewind [72] which defines a framework that consists of six elements (association, consistency, discriminative power, tracking, predictability and repeatability) for validating software quality metrics. With regard to the validation of software security metrics, Verendel stated that the validation of “quantified security” is crucial, and thus it needs to be addressed in order for security metrics to be reliable [73].

Another common approach for validating software metrics is through empirical validation which aims to provide measurable evidence of validity [74], for example, collecting data from a systematic experiment [73]. This approach is carried out by a number of researchers to prove the validity of their proposed metrics including the approaches of Subramanyam and Krishnan [51], Zhou and Leung [75], Olague et al. [76] and Shen et al. [77]. With regard to security, this approach has been widely considered by many researchers in order to validate their security metrics, including the work of Manadhata et al. [41], Ozment [78], Naqvi and Riguidel [79] and Chowdhury and Zulkernine [80].
2.4. Refactoring

Another approach is through theoretical validation which aims to formally prove that certain arguments support the anticipated results [73]. Many software metrics have been validated using this approach, for instance the work of Fenton [81] and Melton et al. [82]. Furthermore, there exist many software security metrics which have been validated theoretically including the works of Sallhammar et al. [83], Singh et al. [84] and Buldas et al. [85].

Even though there are other approaches for validating software metrics such as simulation and theorem proving [73], Kitchenham et al. [74] stated that the most efficient approach for validating software metrics is one which takes into account both empirical and theoretical approaches.

2.3.3 Summary of Previous Security Metrics

It can be seen that most security metrics assess security at either a very high level (i.e., the abstract system’s architecture) or at a fine level of granularity (i.e., with respect to individual program coding constructs). However, the most efficient approach for quantifying overall security of a given object-oriented program is one which defined based on the compositional properties of object-oriented programs (e.g., coupling and cohesion). It also needs to consider data flow analysis principles that trace potential information flow between high- and low-security system variables. In this research we aim to develop a number of security metrics that are capable of measuring overall security of a given object-oriented program based on many of its compositional properties and information flow principles.

2.4 Refactoring

Refactoring is an important aspect of software evolution since it aims to increase the quality of software [86]. It 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 behaviour” [18].

There are two main requirements for refactoring that can be seen from this definition. The first is improving the program’s quality. The second is ensuring that restructuring the program does not change its functional behaviour. There are two types of refactoring [86]: code refactoring, and design refactoring.

Since refactoring steps may change a program or design’s quality, including its security
quality, refactoring is highly relevant to security metrics and can provide a way of validating their correctness. “Good” refactoring steps should be detectable by our security metrics.

Refactoring is similar in a way to performance optimisation since both do not change the functional behaviour of the program. However, performance optimisations often make the code harder to understand [18]. On the other hand, refactoring aims to make the code easier to understand but does not always enhance performance [18].

Using refactoring involves two main activities: (1) adding functions and (2) restructuring those functions [18]. Adding functions involves inserting new tasks without changing the existing code and testing those tasks to ensure that they are implemented correctly [18]. The second activity is to restructure the interactions between these new functions in a better way [18]. Refactoring can be done as many times as needed provided it fulfils its two requirements. It has been shown that the continuous process of refactoring is essential for both the applications and their developers [86]. Fowler [18] has listed 72 refactoring rules in detail. In addition, the refactoring web page includes more refactoring rules [87]. However, none of these mention the impact of refactoring on security.

2.4.1 Refactoring Objectives

One of the objectives of refactoring is to improve the software’s design [18]. Unstructured code modifications can result in the program losing its design and becoming harder to understand [18]. Long-lived software inevitably evolves due to bug fixes and extensions to its functionality, often introducing redundant code that makes the system design hard to understand. However, refactoring aims to eliminate such redundancy [18].

Another important objective of refactoring is to make software easier to understand [18], and thus easier to maintain [18]. Refactored code is also more likely to have a better design or structure [18]. This principle can also be applied to the software’s architectural design.

Fowler claims that refactoring can assist in finding bugs [18]. He states that clarifying the program’s structure leads to a faster way of identifying bugs. Enhancing the program development process is another objective of refactoring [18]. Since refactoring aims to improve software quality, it can then help reduce the time required to implement new functions and fix bugs [18]. All of these issues affect the program’s overall quality and, hence, its security.
2.4.2 The Refactoring activities

![Diagram of the refactoring process]

Figure 2.2: The refactoring process

Figure 2.2 shows how the six typical refactoring activities are carried out. Unfortunately, the standard references on refactoring do not consider secure refactoring, even though these steps can significantly alter a program’s security. Below is a detailed description of each of the refactoring activities.

**Activity One: Identify where to refactor**  As Mens and Touwre state, the first two activities are always combined when refactoring is explained. In general, there are two main decisions that need to be taken when deciding where to apply refactoring.

1. The level of abstraction at which to apply the refactoring, to the program (i.e., code level) or to more abstract software artifacts (such as design models or requirements documents).

2. Where the refactoring is applicable. Two of the ways stated by Mens and Touwre to apply this are:

   (a) Automated tools that detect program weaknesses.

   (b) Identifying “Bad Smells” which Beck defines as “certain structures in the code that suggest the possibility of refactoring.” This can be done in many ways such as
by Simon et al.’s [89] approach (object-oriented metrics) or by Ducasse et al.’s [90] approach (object-oriented meta model). The Logic Meta Programming technique has proven to be excellent for bad smells detection [91].

**Activity Two: Identify refactoring methods**  The second activity is to decide on the methods to be chosen to implement refactoring. This decision is mainly affected by the type of the software, for example, whether it is a web-based application, stand alone application or others [88].

**Activity Three: Ensure to preserve software behaviour**  This activity is mainly concerned with making sure that the software does not lose its original behaviour after applying refactoring. However, this activity is not as easy as it sounds since there is no clear definition of behaviour [88]. Nevertheless, behaviour preservation was first defined by Opdyke who stated that behaviour preservation is ensured only if the output values resulting before and after the refactoring are the same for the same input values [92]. Note that this definition refers to program functionality but not characteristics such as performance or security.

Opdyke has also suggested the refactoring precondition approach to accomplish this definition [92]. Software behaviour preservation varies depending mainly on the application domain [88]. For example real-time software behaviour preservation depends on the execution time of specific operations [88].

Another method to assess behaviour preservation is through testing [88]. This method is mainly described as specifying a set of test cases that should pass before and after the refactoring to ensure that the software has preserved its original behaviour [88]. Unfortunately, the main limitation of this method is that in many cases changing the structure of the program can cause many test cases to fail even if the software behaviour has not been changed [88].

Another approach is to use formal verification principles to prove that the refactoring rules preserve the program’s semantics whenever they’re applied. This is the strongest possible way to guarantee that the rules will do what we want, but verification is a theorem proving activity and can be challenging [93].

There are also a number of other suggested approaches which assess behaviour preservation after refactoring, such as adaptation of a weaker notion of behaviour preservation [88].
However, whichever method is used, behaviour preservation is still very difficult to assess.

**Activity Four: Applying refactoring** The fourth activity is to apply the selected refactoring methods at the selected sections. This is can be done manually or using one of the existing tools, for instance, JRefactory for Java or C# Refactory for C#.NET.

**Activity Five: Assessing software quality characteristics after refactoring** Software has two types of quality entities: internal (coupling and cohesion) and external (usability and performance) [88]. Refactoring methods should be classified by whichever of these entities they mainly affect. This would improve the chance of selecting methods that improve the program’s quality [88]. However, this cannot be done without an analysis of the effect of refactoring methods on software quality based on those two entities [88].

There are a number of defined approaches which help in estimating the effect of refactoring on software quality characteristics. Some of these are software metrics, empirical measurements, controlled experiments, and statistical techniques [88].

**Activity Six: Maintaining consistency after refactoring** During the software development life cycle, many activities are conducted such as requirements gathering, analysing the requirements, designing the software models and so on. Any refactoring of one of these would need to ensure that it is still consistent with the others [88]. Thus, this would require a mechanism or an approach to ensure this consistency. Three approaches have been defined in this area [88].

One is Bottoni et al.’s [94] approach which involves maintaining consistency between the code and the design models. Another is to maintain consistency within the same domain, such as in Van Der Straeten et al.’s [95] approach which is based on logic rules. The third approach is to handle consistency between software artifacts, such as Rajlich’s [96] approach which is called “change propagation”.

**2.4.3 Refactoring and Performance**

There is no doubt that refactoring has an effect on the performance of a program, just like any other changes to a program. However, changes that aim to make the program easier
to understand have always been the key reason for slowing the program’s performance [18]. Fowler [18] argues that refactoring often causes software to slow its performance. However, he states that refactoring “makes the software more amenable to performance tuning” in real time contexts [18]. However, there is one case that has shown that refactoring programs can make performance better as noted by Demeyer [97]. This is when conditional logic is replaced with polymorphism.

To conclude, to optimise software efficiently, it is necessary to have a full understanding of the program. Thus, refactoring makes the program easier to understand which can lead to better optimization results.

2.4.4 Refactoring and Design Quality

Software quality covers many aspects including reliability, portability and maintainability. However, refactoring usually focuses on maintainability as indicated by the refactoring definition [98].

Design has been always a key role in the software development life cycle. It is actually considered the most important factor in software quality measurements [98]. As a result of that, many software designers consider “Upfront Design” [18] which aims to produce the best possible design. This would lead to reducing the cost and time of the software development project. An alternative approach is to code the program and test it. When it is correct, then the program can be refactored [18]. Both of these two approaches will eventually produce a good design of the system, and can be used together [18].

An experiment carried out by Stroggylos and Spinellis [98] showed that refactoring can lead to less coherent classes in object-oriented systems and increased complexity metrics in procedural systems. The experiment concluded that either the refactoring process does not always improve software quality, or the developers still have not found the most efficient way of applying refactoring to improve software quality [98]. Another study by Bois et al. [99] analysed the impact of refactoring on cohesion and coupling which are considered to be key factors on software maintainability. The study showed that it is possible to improve the cohesion and coupling of a system with a specific set of refactorings, therefore, improving the software quality overall [99].
2.4.5 Refactoring and Security

Refactoring is an important aspect of software evolution since it aims to increase the quality of software [86]. However, its impact on software security was hardly mentioned or studied until recently [58]. Maruyama and Tokoda [100] 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 [58]. The author proposed four refactoring steps for Java source code which aim to protect the confidentiality of secret data in a given program. These rules consist of Introduce Immutable Field which declares fields as ‘final’ if their value is only set once [58]. Another rule aims to protect the secrecy of confidential fields through the rule of Replace Reference with Copy [58]. The other rules include Prohibit Overriding which prevents sensitive methods from being overridden and Clear Sensitive Value Explicitly which allows us to delete sensitive data from the program’s memory as early as possible [58].

Furthermore, Smith and Thober [101] have identified a refactoring approach for critical systems similar to the approach of Li and Zdancewic [102]. Both of these approaches aim to refactor a program’s code into two modules; a high-security and a low-security one. Smith and Thober admit that this is a very challenging task as many real programs share others’ libraries and code between them. Therefore, detecting which classes are high security and which ones are not is, in many cases, very difficult [101].

Another work in the area of secure refactoring is Hafiz’s [103] which defines a number of secure transformation rules. These rules aim to refactor program code in order to change its functionality to prevent well-known kinds of security vulnerability such as buffer overruns, code injection attacks, lack of access control and poor isolation. However, this approach does not consider the potential flow of classified information within a given program but instead aims to avoid well-known coding errors.

However, although many of these approaches claim to improve the security of a given program by removing existing vulnerabilities, they do not guarantee that new vulnerabilities
will never be introduced. Additionally, they do not quantify the impact of changes on the overall security level of a given program, and they require full source code implementations of the programs, which is inevitably less efficient than finding problems at design time.

There are a number of identified software metrics which can be used to detect software weaknesses which require refactoring, for example, Joshi and Joshi’s approach \cite{104} but these do not include security metrics. Existing refactoring rules will often have an impact on security for all programs. For example the *Encapsulate Field* refactoring rule by Fowler \cite{18} may improve security by hiding fields which contain secret data from public access. Similarly, the refactoring rule of *Encapsulate Class with Factory* by Kerievsky \cite{105} may hide classes which contain confidential data from public access. As a result of this, our security metrics are beneficial when identifying potential refactoring rules, selecting the rule (and parameters) to be applied, applying the rules, and assessing their effect.

### 2.4.6 Significance of Security Metrics for Refactoring

Given that refactoring may change program code’s security level, security metrics are needed to show the impact of refactoring on a program’s security. Such metrics will help in assessing the impact of existing refactoring rules on the security of programs in a number of ways. One way is that security design metrics will help in measuring how design-level refactoring rules may affect security. The other way is to assess the impact of code-level refactoring rules on security. These two approaches will help to define a set of security-aware refactoring rules for the design and code levels that guarantee to improve (or at least not worsen) security at these levels.

### 2.5 Discussion

Our literature review has been in three main parts: one is related to software security design, another has discussed refactoring and the third has illustrated some of the existing work on software quality measurements including existing software security metrics. These can be used as guidance for future secure software development processes and as elements of a secure systems architecture. Many of the studies in this area focus on security at the level of individual program coding constructs. These approaches, which are related to security, intended as either guidance to help develop more secure systems or measure the security of individual program
statements. They are not capable of quantifying the security of a given program either from its
design or code level. Furthermore, they cannot detect the change to security when programs are
refactored. Thus, there is a need for security metrics which objectively measure the security of
various programs from either the design or code levels. Such metrics will also be capable of
assessing the impact of refactoring on a program’s security from various levels, thus making the
metrics useful during system design, coding and maintenance.

Our project differs from previous work as it focuses on the security of the overall program
module structure. We study various areas related to the security of object-oriented systems
(such as, architecture, design principles, and refactoring) in order to achieve the main goal
helping introduce a good security design into an existing program.

A major outcome of our work is software security metrics for assessing the security level of
an existing program. Other outcomes include security notations and identification of refactoring
rules which guarantee to introduce improvements in a given program’s security.

A major challenge for our project is how to validate the outcomes. A study conducted
by Tempero [106] looked at over 100 open source Java programs to see to what extent a given
program declared non-private attributes but does not use them. The study found out that up to 87
programs have at least one class with public fields. The most surprising result was that it is very
common for real applications to declare a non-private field and not use it later on [106]. This
result is consistent with the author’s previous studies concerned with to what extent software
design decisions are not followed [107] and also consistent with another study which confirmed
that many interfaces in programs are not implemented [108]. Having seen the value of such
an empirical approach, we also analyse suits of open-source software to validate our security
metrics.
Part II

OBJECT-ORIENTED DESIGN-LEVEL SECURITY ASSESSMENT
In this chapter we focus on the security design of individual classes and define a number of security metrics for UML-like class designs. The metrics assess the ways which the design grants access to ‘classified’ attributes, allow designers to discover and fix security vulnerabilities at an early stage, and help compare the security of various alternative designs. In particular, we present seven security metrics to measure Data Encapsulation (accessibility) and Cohesion (intra-class interaction) of a given object-oriented class from the point of view of potential information flow. Information flow is a simple, intuitive, system-independent measure of data confidentiality and integrity.

3.1 Introduction

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 \[2, 3\]. Several projects have been conducted to investigate information flow through computer program code. This has been studied using several approaches, including type analysis \[9\] and data/control-flow analysis \[10\].

However, 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 \[33\]. These
principles were intended as guidance to help develop secure systems, mainly operating systems. Bishop’s [32] and McGraw’s [11] 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 [3, 58]. Security measurements have been defined to assess security at the system level [41] and the level of implementation code [63]. However, measuring security at the design phase, based on typical design artifacts, has received little attention.

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]. In particular, the National Institute of Standards and Technology [109] stated that eliminating vulnerabilities in the design stage can cost 30 times less than fixing them at a later stage. Therefore, defining a set of metrics which evaluates the security of a given program based on its design rather than its source code would reduce the cost of fixing security design vulnerabilities by detecting these vulnerabilities at an early stage. This chapter presents a new set of metrics which are capable of assessing the security quality of object-oriented classes from an information flow perspective. They can be used to compare different designs for the same class and identify the best design for a certain security design principle.

We base our metrics on the quality properties specified by Bansiya and Davis [56] as this model clearly defines most of the program’s quality properties and how high level properties such as coupling and cohesion are related to lower level metrics. We have chosen properties which are relevant to individual object-oriented classes: data encapsulation and cohesion. Our 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 [28] while SPARK’s annotations express the information flow relations between methods and attributes of a given class [17], 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 for the class. This can be done by either comparing the overall results or by choosing those results relevant to a certain security design principle.

### 3.2 Assumptions and Annotations

The security design metrics in this chapter are capable of quantifying the security level of a single object-oriented class. They are different from typical “code complexity” metrics, which measure syntactic properties of the program 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, with lower values indicating that the program’s design is more secure. (Thus the metrics can be considered to measure the insecurity of a design.) Their results show which alternative designs will improve or worsen the security of a given class with regard to a specific software security design principle (e.g., Least Privilege, Reduce Attack Surface, etc. [32, 40, 43]).

The metrics presented in this chapter are concerned with information flow through individual object-oriented classes. Two properties are covered: the accessibility of classified attributes, and interactions between classified and non-classified attributes inside a given class. (Other properties of multiple classes, such as inheritance, coupling, and extensibility, are considered in the following chapter.)

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 (shown in Figure 3.1). UMLsec [28] 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 [28]. Our metrics consider classified data as that which is labelled by “secrecy” in UMLsec. For our purposes, we adopt two annotations of UMLsec, specifically ‘secrecy’ to annotate attributes which include classified data and ‘critical’ for annotating classes that contain attributes annotated as ‘secrecy’.

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 [17]. 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 (shown in Figure [3.1]) consist of a “derives from” block which explains how the value of a certain variable or method’s return value is (potentially) derived from the value of another method parameter or variable [17].

Some terminology associated with our metrics is defined as follows.

- **Classified Attribute**: An attribute which is defined in UMLsec as ‘secrecy’.
- **Instance Attribute**: An attribute whose value is stored separately by each instance of a class [110, 111].
- **Class Attribute**: An attribute whose value is shared by all instances of that class [110, 111].
- **Classified Method**: A method which reads or writes at least one classified attribute.
- **Unclassified Method**: A method which does not interact with any classified attributes.
- **Mutator**: A method that sets the value of an attribute [110, 111].
- **Accessor**: A method that reads the value of an attribute [110, 111].
- **Classified Mutator**: A method that sets the value of at least one classified attribute.
- **Classified Accessor**: A method that reads the value of at least one classified attribute.

In order to measure the impact of these properties on information flow, we rely on the programmer to provide us with security-annotated class diagrams. We assume the programmer correctly annotates all sources of classified data in their design/code and that our subsequent analysis of where classified data may propagate to is dependent on the accuracy of the programmer’s annotations.

### 3.3 Relevant Security Design Principles

This section describes the security design principles covered by our security metrics for individual classes. As mentioned previously, a number of studies have presented several design
principles for developing secure systems [33, 32, 11]. In this chapter, we have chosen two
principles to measure the security of designs from the perspective of information flow: least
privilege [32] and reduce attack surface [40]. These two principles are those which have the
greatest relevance to the flow of data through program code.

The principle of least privilege means “programs and users should run with the least
privilege to complete their job” [33]. The main advantage of this principle is to minimise
the interactions among privileged programs [33]. 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 [40] 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 interact with classified attributes needed for necessary tasks. A class
which has the least necessary accessibility to classified attributes would satisfy the requirements
of the security principle of reducing the attack surface.

3.4 Security Design Metrics

This section defines and illustrates our security design metrics. These metrics aim to measure
the relative size of the attack surface and the amount of privilege granted to parts of the program,
so counting vulnerabilities is inevitably a basic aspect of the approach. We divide our security
design metrics into two categories. One is related to accessibility while the other is concerned
with interactions. These metrics are defined based on an analysis of software quality design
metrics defined in the Quality Model for Object-Oriented Design [56] which uses ratios of
cardinalities to calculate their metrics.
3.4.1 Security Accessibility Metrics

Metrics under this category aim to measure the accessibility 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 [110]. These modifiers correspond to the Java keywords which include: public, protected, and private.

Maruyama and Tokoda [100] 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 change its security level. However, this approach does not 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 [55] 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 [55].

Bansiya [55] 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 [55].

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 by the designer as classified since they are the ones which need to be kept secret. We divide the metrics for individual classes into three kinds of accessibility: instance attributes; class attributes; and methods.

**Classified Instance Data Accessibility (CIDA)** This metric measures the direct accessibility of classified instance attributes of a particular class. It helps to protect the classified internal representations of a class, i.e., instance attributes, from direct access. It is defined as “The ratio of the number of non-private classified instance attributes to the number of classified attributes in a class”. Therefore, it is calculated by dividing the number of non-private classified instance attributes (such as attribute areaCode in Figure 3.1) in a class to its total number of classified attributes. This gives us the ratio of classified instance attributes which have direct access from outside the class. Higher values indicate higher accessibility to these classified attributes and hence a larger ‘attack surface’. This means a higher possibility for confidential data to be
exposed to unauthorised parties. Aiming for lower values of this metric adheres to the security principle of reducing the attack surface [40].

Consider the set of classified attributes in class $C$ to be $CA = \{ca_1, \ldots, ca_n\}$ and its non-private classified instance attributes as $NCIA = \{ncia_1, \ldots, ncia_n\}$ such that $NCIA \subseteq CA$. For an arbitrary set $S$ let $|S|$ denote its magnitude, i.e., the number of elements it contains. Then:

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

**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 non-private classified class attributes to the number of classified attributes in a class”. This metric is calculated by dividing the number of non-private 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 the class. Higher values mean that confidential data of that class has a higher chance of being exposed to unauthorised parties. This metric contributes towards measuring the attack surface size of a given program’s class classified attributes. Thus, lower values of this metric enforce the security principle of reducing the attack surface [40].

Consider the set of classified attributes in class $C$ as $CA = \{ca_1, \ldots, ca_n\}$ and the non-private classified class attributes as $NCCA = \{ncca_1, \ldots, ncca_n\}$ such that $NCCA \subseteq CA$. Then:

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

**Classified Operation Accessibility (COA)** This metric is the ratio of the accessibility of non-private classified methods of a particular class. We define it as: “The ratio of the number of non-private classified methods to the number of classified methods in a class”. It is calculated by dividing the number of classified methods which are not declared as private (such as method
GetArea in Figure 3.1) 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 non-private methods. This metric measures the potential attack surface size exposed by classified methods [40].

Consider the set of all classified methods in class $C$ as $CM = \{cm_1, \ldots, cm_n\}$ and the non-private classified methods in that class as $NCM = \{ncm_1, \ldots, ncm_n\}$ such that $NCM \subseteq CM$. Then:

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

(3.3)

3.4.2 Security Interactions Metrics

Our interactions metrics are defined to measure the impact of class interactions (intra-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. [53]. 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 [53].

However, our interaction metrics instead measure the flow of information caused by methods’ and attributes’ interactions in a given class according to the designer’s annotations. 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 accessors with classified attributes; the weight of classified attributes interactions; and the weight of classified methods.

Classified Mutator Attribute Interactions (CMAI) This metric measures the interactions of mutators with classified attributes in a class (such classified mutator include method SetArea in Figure 3.1 which mutates classified attribute areaCode). We define it 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 from how many places in the design/program classified attributes may be mutated (such as mutator method of SetArea in Figure 3.2). Then, we divide this number by the total number of possible ways of mutating these classified attributes via any method within their scope. The result is a ratio which can be used 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 [32].

Consider a set of mutator methods in class \( C \) as \( MM_i, i \in \{1, \ldots, mm\} \) and the classified attributes as \( CA_j, j \in \{1, \ldots, ca\} \). Let \( \alpha(CA_j) \) be the number of mutator methods which may access classified attribute \( CA_j \) according to the programmer’s annotations. Integer product \( |MM| \times |CA| \) then represents all the ways methods in the class could potentially interact with classified attributes. 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 accessors with classified attributes in a class (such classified accessor include method GetArea in Figure 3.1 which accesses classified attribute areaCode). We define it as: “The ratio of the number of accessors which may interact with classified attributes to the possible maximum number of accessors 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 according to the programmer’s annotations. Then, this number is divided by the total number of possible ways of accessing these classified attributes by any method within their scope. This results in a ratio which directly shows the actual versus 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, and 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 satisfies the security principle of least privilege [32].

Consider a set of accessor methods in class $C$ as $AM_i, i \in \{1, \ldots, am\}$ and classified attributes as $CA_j, j \in \{1, \ldots, ca\}$. Let $\beta(\text{CA}_j)$ be the number of accessor methods which may access attribute $\text{CA}_j$ according to the programmer’s annotations. Integer product $|AM| \times |CA|$ is the number of ways any accessor could read a classified attribute. Then, $CAAI$ for class $C$ can be calculated as:

$$CAAI(C) = \sum_{j=1}^{ca} \frac{\beta(\text{CA}_j)}{|AM| \times |CA|}$$  \hspace{1cm} (3.5)

**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 both as per the programmer’s annotations. 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 [32].

Consider a set of attributes in class $C$ as $A_i, i \in \{1, \ldots, a\}$ and a set of classified attributes as $CA_j, j \in \{1, \ldots, ca\}$. Let $\gamma(\text{CA}_j)$ be the number of methods which may access classified attribute $\text{CA}_j$. Let $\delta(A_i)$ be the number of methods which may access attribute $A_i$. Both functions are calculated using the relationships between methods and attributes expressed by the programmer’s annotations. Then, $CAIW$ can be computed as:

$$CAIW(C) = \frac{\sum_{j=1}^{ca} \gamma(\text{CA}_j)}{\sum_{i=1}^{a} \delta(A_i)}$$  \hspace{1cm} (3.6)
3.5 Individual-Class Design Metrics Case Study

**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 according to the programmer’s annotations. 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 does not focus on accessibility but instead it focuses on the interaction weight of classified methods. Higher values of this metric indicate that more privileges over classified data are offered by the given class. This leads to a higher chance of information flow of classified data by the privileges granted to the class’s methods and violations of the security principle of least privilege [32].

Consider a set of methods in a class $C$ as $M = \{m_1, \ldots, m_n\}$ and the classified methods as $CM = \{cm_1, \ldots, cm_n\}$ such that $CM \subseteq M$. Then, $CMW$ is expressed as:

$$CMW(C) = \frac{|CM|}{|M|} \quad (3.7)$$

### 3.5 Individual-Class Design Metrics Case Study

The following case study illustrates how our security class design metrics are used. They can be applied once a UML class diagram of a single class is constructed. The 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 their metrics. There are a number of assumptions associated with these metrics:

- Any method that changes the value of an attribute (according to the annotations) is a mutator.

- Any method that reads the value of an attribute (according to the annotations) 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 change the value of any attributes.

- Some object-oriented programming languages allow methods to have parameters which return values (such as “out parameters” in C#); we consider these methods as accessors.

### 3.5.1 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. Note the annotations which indicate which methods are classified and how values are exchanged between class attributes and method parameters.

![Figure 3.1: ContactNos 1 class diagram](image-url)

```
+ name : String
+ secrecy areaCode : String
+ secrecy officeNo : String

+ SetName(_name : String) : void
  [derives name from _name]
+ GetName() : String
  [derives GetName() from name]
+ SetTelephone(_area : String , _office : String) : void
  [derives areaCode , officeNo from _area , _office]
+ SetArea(_area : String) : void
  [derives areaCode from _area]
+ SetOffice(_office : String) : void
  [derives officeNo from _office]
+ GetTelephone() : String
  [derives GetTelephone() from areaCode , officeNo]
+ GetArea() : String
  [derives GetArea() from areaCode]
+ GetOffice() : String
  [derives GetOffice() from officeNo]
```
To illustrate the capabilities of our metrics, we apply them to seven refactored versions of the ContactNos design. We assume that Figure 3.1 shows the original ContactNos UML class diagram. Figures 3.2 to 3.7 show refactored versions of the original design using one or more the refactoring methods defined by Fowler [18, 87]. These proceeding refactoring diagrams are independent of each other but they are a different version of the initial program. (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 chapter. Designs that use more than one class can be found in Chapter 4.)

Figure 3.2 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 3.1 in which they were public. This is done by using the Encapsulate Field refactoring rule [18, 87]. 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 3.2 keeps the rest of the methods in Figure 3.1 unchanged except for declaring them as private by using the Hide Method refactoring rule.
Another design is Figure 3.3 which has the same changes as in Figure 3.2. 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 3.4 shows a design which is similar to Figure 3.3 except it has used the refactoring rule Extract Method. Extract Method has been applied to the mutators and accessors of the areaCode and officeNo attributes. This was done to mutate and access their values separately while making their methods private.

Figure 3.5 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 3.6 has the same changes as in Figure 3.5 but has also combined both of the telephone number attributes to one attribute called teleNo which is declared as “secrecy”.

<table>
<thead>
<tr>
<th>«Critical»</th>
<th>ContactNos_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>- name : String</td>
<td></td>
</tr>
<tr>
<td>- «secrecy» areaCode : String</td>
<td></td>
</tr>
<tr>
<td>- «secrecy» officeNo : String</td>
<td></td>
</tr>
<tr>
<td>+ SetDetails(_name : String, _area : Integer, _office : Integer) : void</td>
<td></td>
</tr>
<tr>
<td>[derives name, areaCode, officeNo from _name, _area, _office]</td>
<td></td>
</tr>
<tr>
<td>+ GetDetails() : String [ ]</td>
<td></td>
</tr>
<tr>
<td>[derives GetDetails() from name, areaCode, officeNo]</td>
<td></td>
</tr>
<tr>
<td>- SetName(_name : String) : void</td>
<td></td>
</tr>
<tr>
<td>[derives name from _name]</td>
<td></td>
</tr>
<tr>
<td>- GetName() : String</td>
<td></td>
</tr>
<tr>
<td>[derives GetName() from name]</td>
<td></td>
</tr>
<tr>
<td>- SetArea(_area : Integer) : void</td>
<td></td>
</tr>
<tr>
<td>[derives areaCode from _area]</td>
<td></td>
</tr>
<tr>
<td>- SetOffice(_office : Integer) : void</td>
<td></td>
</tr>
<tr>
<td>[derives officeNo from _office]</td>
<td></td>
</tr>
<tr>
<td>- GetArea() : String</td>
<td></td>
</tr>
<tr>
<td>[derives GetArea() from areaCode]</td>
<td></td>
</tr>
<tr>
<td>- GetOffice() : String</td>
<td></td>
</tr>
<tr>
<td>[derives GetOffice() from officeNo]</td>
<td></td>
</tr>
<tr>
<td>- ParseIntToString(_int : Integer) : String</td>
<td></td>
</tr>
<tr>
<td>[derives ParseIntToString(...) from _int]</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.3: ContactNos 3 class diagram
3.5. **Individual-Class Design Metrics Case Study**

```plaintext

``Critical``

ContactNos_4

- name : String
- «secrecy» areaCode : String
- «secrecy» officeNo : String

+ SetName(_name : String) : void
  [derives name from _name]
+ GetName() : String
  [derives GetName() from name]
+ SetTelephone(_area : Integer , _office : Integer) : void
  [derives areaCode, officeNo from _area, _office]
+ GetTelephone() : String
  [derives GetTelephone() from areaCode, officeNo]
- SetArea(_area : Integer) : void
  [derives areaCode from _area]
- SetOffice(_office : Integer) : void
  [derives officeNo from _office]
- GetArea() : String
  [derives GetArea() from areaCode]
- GetOffice() : String
  [derives GetOffice() from officeNo]
- ParseIntToString(_int : Integer) : String
  [derives ParseIntToString(...) from _int]

``Critical``

ContactNos_5

- name : String
- «secrecy» areaCode : String
- «secrecy» officeNo : String

+ SetDetails(_name : String , _area : String , _office : String) : void
  [derives name, areaCode, officeNo from _name, _area, _office]
+ GetDetails() : String []
  [derives GetDetails() from name, areaCode, officeNo]

``Figure 3.4: ContactNos 4 class diagram``

``Figure 3.5: ContactNos 5 class diagram``

This refactoring rule is not shown in the literature but we call it "**Inline Field**".

**Figure 3.7** 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.
Chapter 3. Security Metrics for Object-Oriented Class Designs

Table 3.1 shows the results of applying our metrics to the seven class designs shown above. To make it easier to understand and compare these results, we also show them as radar charts in Figures 3.8 to 3.11. This allows us to easily compare which aspects of the system are most secure and which ones are not. Given that lower values 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 3.8 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 do not have any classified class attributes.

In terms of classified methods accessibility, the class diagrams of ContactNos 1, 5, 6, and 7...
3.5. Individual-Class Design Metrics Case Study

![Figure 3.8: ContactNos 1 metrics results](image1)

![Figure 3.9: ContactNos 2 versus ContactNos 3](image2)
Figure 3.10: ContactNos 4 versus ContactNos 5

Figure 3.11: ContactNos 6 versus ContactNos 7
Table 3.1: 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>0.5</td>
<td>0.5</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>

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 and 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, 3, 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 secure 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.
3.5.3 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. (In Chapter 7 we also show how our metrics can be summarised to form a hierarchy.)

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 3.3: 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 CMAI, CAAI, CAIW and CMW metrics have yielded the lowest value. On the other hand, from the requirement of reducing the attack surface the designs of ContactNos 2, 3 and 4 are the most secure because the CIDA, CCDA and COA metrics have produced the lowest values.

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

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

3.6 Conclusion

In this chapter, we have defined a number of security metrics for individual object-oriented class designs. These metrics are easy to calculate 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 class design but they can also give indications of where any potential vulnerability occurs since they measure various ways of potential flow of classified data. They differ from code level metrics as they are easier to capture and do not require the software to be implemented. We have
also shown how to directly compare the metrics results for various alternative designs of a class and thus help choose the design which best satisfies a certain security design principle. In the next chapter, we define another suite of security metrics which cover the entire design of a multi-class object-oriented system.
Security Metrics for Multi-Class Object-Oriented Designs

In this chapter we extend our individual-class security metrics from the previous chapter to focus on the entire design of an object-oriented application. We measure the compositional properties of security-annotated object-oriented designs to help software designers compare the potential security of various alternative designs. In particular, we present security metrics based on composition, coupling, extensibility, inheritance, and the design size of a given object-oriented, multi-class program from the point of view of potential information flow.

4.1 Introduction

This chapter extends our individual-class security metrics from the previous chapter and proposes a new set of metrics which are capable of assessing information-flow security of whole object-oriented designs. In Chapter 3 we developed seven security metrics for assessing the security of a single object-oriented class. These metrics measured Data Encapsulation and Cohesion for a given class. Here we extend that work to consider entire class hierarchies.

We define our metrics based on the quality properties for object-oriented programs specified by Bansiya and Davis [56], using five properties which are related to the overall design of an object-oriented program: composition, coupling, extensibility, inheritance, and design size. Our metrics aim to measure any potential information flow which could occur between objects instantiated from the design’s classes with regard to the security design principles of “reducing
the size of the attack surface” [40] and “least privilege” [33, 32].

However, in order to measure the impact of these properties on information flow, we again need security-annotated class diagrams. Similar to our single class metrics, we use UMLsec and SPARK’s annotations to identify confidential data [28] and to express the information flow relations between attributes, methods, and classes of a given design [17]. Once the metrics’ results are identified for a number of alternative designs, it is then easy to choose the most secure design.

4.2 Metrics Development and Definitions

We developed our multi-class security metrics based on an analysis of software quality design properties defined in the Quality Model for Object-Oriented Design [56]. These properties include: composition, coupling, extensibility, inheritance, and design size. We studied each property and its relevance to designing secure software to define our metrics so that if any security-critical properties have changed in a design then the relevant metrics will reflect this change as well.

4.2.1 Composition

Aggregation and composition are extensively mentioned in the context of object-oriented programming. Aggregation is an association between two or more system components/objects [112, 113]. It has been described as a ‘whole-part association’, where one object is the whole and the others are the parts [112]. However, composition is a stronger type of aggregation which has a ‘lifetime dependency’ between composite objects and the whole object [112]. Therefore, deleting the whole object would result in deleting the composite ones [112]. In object-oriented programming, composition is implemented through the use of inner/nested classes [114], by implementing the composite whole class as the outer class and the composite part class as the inner one. It is expected that no other class would have access to the inner class directly, therefore, access to inner classes is done by first having access to the outer class [13].

However, this is not always true in program code since some programming languages treat inner classes as independent ones, and therefore allow direct access to them [13]. However,
at the design level, we assume this cannot occur and that the design describes the intended implementation accurately. Such implementation-dependent issues are beyond the scope of design-time analyses.

Composition yields a weak possibility of potential information flow for classified data when considering information security. This risk has been identified in the field of software security, and as a result it is recommended to avoid using inner classes in security-critical code [13]. However, in our case we assume that using inner classes is secure since most programming languages do not allow external access to inner classes, unless they are marked as public.

![Figure 4.1: Composition Class Hierarchy](image)

Figure 4.1 shows a class hierarchy which has employed composition. The design includes two classified data attributes, $\delta$ of type $U$ and $\gamma$ of type $V$, distributed among two classes: class $A$ (outer class) and class $B$ (inner class). We use Greek letters in these examples to denote classified data. (In UMLsec they would be labelled “secrecy”.) Our composition metric aims to promote the use of critical composite parts which contain classified data similar to class $B$ and discourage the use of composite whole classes similar to class $A$. This will reduce potential information flow of classified data from composite-whole to composite-part classes.

**Composite-Part Critical Classes (CPCC)** This metric is defined as “The ratio of the number of critical composed-part classes to the total number of critical classes in a design”. It aims to reward the use of inner classes for holding classified data, and penalise the use of outer classes for this purpose. We assume that in order to access inner classes, it is essential to access
the outer class, and therefore no direct access can be made to the inner class \[13\]. From this point of view, storing private classified data in inner classes is a more secure solution since this data has less chance to be exposed to the public. Increasing the number of critical composed-part classes in a design gives a lower value of this metric, and hence indicates a more secure design. Aiming for lower values of this metric adheres to the requirement of the security design principle of “*reducing the size of the attack surface*” \[40\].

Consider a set of critical classes in a design \(D\) as \(CC = \{cc_1, \ldots, cc_n\}\) and the composed-part critical classes in the same design as \(CP = \{cp_1, \ldots, cp_n\}\) such that \(CP \subseteq CC\). Then, we define the Composite-Part Critical Classes metric as follows, where the magnitude operator \(|S|\) returns the size of a set \(S\).

\[
CPCC(D) = 1 - \left( \frac{|CP|}{|CC|} \right)
\] (4.1)

### 4.2.2 Coupling

Coupling is one of the most important software design properties. It is defined as the degree of interaction an object has with other objects, measured by the number of links it has with these objects \[112, 113\]. A number of metrics have been defined to measure the coupling between classes of an object-oriented program \[45, 54\]. A design time coupling metric is defined by Bansiya and Davis \[56\] called Direct Class Coupling (DCC). Their metric measures the number of other classes that a certain class interacts with \[56\]. A system with low coupling is considered a better design with regard to reusability, understandability, and extensibility \[56, 112\].

The impact of coupling on security has been discussed by Liu and Traore \[115\] in their study which shows a strong correlation between coupling and a system’s attackability. Systems with high coupling are a greater target for successful attacks unlike systems with low coupling \[115\]. With regard to security coupling metrics, a study conducted by Chowdhury et al. \[63\] measures the security coupling between a program’s methods based on code inspections. Our security coupling metric can be applied based solely on the *design* of a given program.

**Critical Classes Coupling (CCC)** This metric measures the degree of security-relevant coupling between classes and classified attributes in a given design. It is defined as “*The ratio of the number of all classes’ links with classified attributes to the total number of possible links*
4.2. Metrics Development and Definitions

with classified attributes in a given design”. It is calculated based on the theory of directed weighted links. In our context, a directed link shows the classes which may reference or access certain classified attributes in other classes. For each class that may access a classified attribute (as indicated by one or more ‘derives from’ annotations), we add a weight of one. To find the ratio of the coupling, we need to first identify the sum of the classified attributes’ accessing or referencing weights. Then, we divide this number by the total number of links to all classified attributes, which is the total number of classes, less the classified attribute’s own class, times the total number of classified attributes.

This metric aims to penalise programs with high coupling. Therefore, it produces lower values for fewer interactions between classes and classified attributes, and hence a lower chance of potential flow of classified data which satisfies the security design principle of “least privilege” [33, 32].

Figure 4.2 illustrates one way of coupling measured by this metric, through the calling of methods in other classes which return classified data. In this case, objects of class C call method M in class B which returns the value of classified attribute \( \delta \). Another way of coupling is through the use of public classified attributes as in Figure 4.3. In this case, objects of class C directly read the value of classified attribute \( \delta \) in class B since it is declared public.

Consider a set of classes in a design \( D \) as \( C_i, i \in \{1, \ldots, c\} \) and a set of classified attributes as \( CA_j, j \in \{1, \ldots, ca\} \), and let \( \alpha(CA_j) \) be the number of classes which interact with classified attribute \( CA_j \) according to the programmer’s annotations. Then, the CCC metric for design \( D \) can be expressed as follows. (A given metric cannot produce a negative number
since a program needs to have at least one construct of interest to exist such as a class or method. In situations where the program does not have the relevant constructs or features of interest for a particular metric, which would produce a zero denominator in the metric, the whole metric is treated as zero.)

\[
CCC(D) = \frac{\sum_{j=1}^{ca} \alpha (CA_j)}{|C| - 1 \times |CA|}
\] (4.2)

4.2.3 Extensibility

Extensibility is the property which allows a certain class or method to be extended by other classes or methods [112, 113]. McGraw and Felten’s text [13] identifies twelve rules for developing more secure Java code. One of these rules is the necessity of preventing classes and methods from being extended. The text mentions that extensibility is an enemy of secure code, and therefore it is essential to make classes and methods inextensible (finalised) unless there is a convincing reason not to do so [13]. With regard to this security coding rule, we have identified two metrics which measure how it is possible to extend the system’s classes, methods, and attributes from a design diagram perspective. The metrics are divided into two kinds: one measures the extensibility of critical classes while the other measures the extensibility of classified methods.

Figure 4.4 illustrates the concept of extensibility. It shows that classes A, B and E are considered critical in this design since they contain classified data denoted by Greek letters. It also shows that class E and method N in class B are not extensible. However, other classes and methods could be extended such as class B and method M in class A.

**Critical Classes Extensibility (CCE)** This metric is defined as “The ratio of the number of the non-finalised critical classes in a design to the total number of critical classes in that design”. Since extensible critical classes could allow other classes to have access to their classified data, their presence increases the “attack surface size” [40]. Making critical classes inextensible eliminates this risk, and hence reduces the possibility of information flow from these classes, so the CCE metric penalises designs with extensible critical classes, such as classes A and B in Figure 4.4. Considering the design in Figure 4.4, the CCE metric would give us a value of two non-finalised critical classes, i.e. class A and B, over three critical classes A, B and E in that
4.2. Metrics Development and Definitions

Consider a set of critical classes in a design $D$ as $CC = \{cc_1, \ldots, cc_n\}$ and extensible critical classes in the same design as $ECC = \{ecc_1, \ldots, ecc_n\}$ such that $ECC \subseteq CC$. Then, we define the Critical Classes Extensibility metric as follows.

$$CCE(D) = \frac{|ECC|}{|CC|} \quad (4.3)$$

**Classified Methods Extensibility (CME)** This metric measures the proportion of non-finalised classified methods which can be extended against the total number of classified methods in a design. We define it as “The ratio of the number of the non-finalised classified methods in a design to the total number of classified methods in that design”. Non-finalised methods allow other methods in other classes to override them. This is a security risk if such methods are classified, and hence the presence of such methods increases the “attack surface size” [40]. This can be avoided by declaring classified methods as inextensible (final), and hence the risk of losing control over classified data can be reduced. Therefore, this metric rewards designs with inextensible classified methods, similar to method $N$ in class $B$ from Figure 4.4.
If we apply the CME metric to Figure 4.4, we get two non-finalised classified methods, i.e. method $M$ in classes $A$ and $E$ respectively over three classified methods $A \cdot M$, $E \cdot M$ and $B \cdot N$ in the whole design which means two thirds of the classified methods in the design are not finalised.

Consider a set of classified methods in a design $D$ as $CM = \{cm_1, \ldots, cm_n\}$ and extensible classified methods in the same design as $ECM = \{ecm_1, \ldots, ecm_n\}$ such that $ECM \subseteq CM$. Then, we define the Classified Methods Extensibility metric as follows.

$$\text{CME}(D) = \frac{|ECM|}{|CM|} \quad (4.4)$$

### 4.2.4 Inheritance

Inheritance is an object-oriented mechanism which allows programmers to provide classes with generalisation and specialisation relationships [112, 113]. Subclasses in an inheritance hierarchy automatically obtain features of their superclasses [112]. The work of Lorenz and Kidd [46] in 1994 aimed to measure inheritance via the number of inherited methods, number of overridden methods, and number of new methods in a class [46]. Bansiya and Davis [56] defined inheritance as the degree of reuse, measured by finding the ratio of the number of methods inherited by a class to the total number of methods accessed by member methods of the class.

Both of these sets of metrics can be measured from a design perspective. However, measuring how inheritance could have an impact on security has not been discussed previously. From the security point of view, inheritance could allow subclasses to acquire privileges over classified data in superclasses, which increases the chance of potential classified information flow. As a result, we have grouped our metrics into four types which cover various kinds of information flow in an inheritance hierarchy.

We illustrate these metrics with regard to Figure 4.5 which shows a class diagram with inherited classified data. Classes $A$, $B$, and $E$ are superclasses, and therefore their public and protected attributes and methods, such as protected method $N$ in class $B$, can be inherited by their subclasses, i.e., classes $F$, $G$, and $I$. However, attributes declared as private in a superclass cannot be inherited such as classified attributes $\delta$ and $\gamma$ in classes $A$ and $B$ respectively.
Critical Superclasses Proportion (CSP)  This inheritance metric measures the proportion of critical superclasses in an inheritance hierarchy. It is defined as “The ratio of the number of critical superclasses to the total number of critical classes in an inheritance hierarchy”. The metric identifies the proportion of critical classes that contain classified data which could be accessed by subclasses. It rewards a lower overall proportion of critical superclasses in a design, and penalises use of critical superclasses, which complies with the requirement of the security design principle of “reducing the attack surface size” [40]. In Figure 4.5 there are two critical superclasses, i.e., classes A and B, compared to four critical classes A, B, F and I in the whole hierarchy.

Consider a set of critical classes in hierarchy $H$ as $CC = \{cc_1, \ldots, cc_n\}$ and critical superclasses in the same hierarchy as $CSC = \{csc_1, \ldots, csc_n\}$ such that $CSC \subseteq CC$. Then, we define the Critical Superclasses Proportion as follows.

$$CSP(H) = \frac{|CSC|}{|CC|} \quad (4.5)$$
Critical Superclasses Inheritance (CSI)  This metric is defined as “The ratio of the sum of classes which inherit from each critical superclass to the number of possible inheritances from all critical classes in a class hierarchy”. This metric penalises class hierarchies in which critical classes appear near the top, and rewards those in which critical classes appear near the bottom, which satisfies the security design principle of “least privilege” [33, 32]. Lower values of this metric indicate that fewer classes may inherit from each critical superclass, and hence there is a less chance of information flow to subclasses. This also means that critical superclasses are towards the bottom of the hierarchy, so these classes are easier to secure [40] because they can have fewer inheriting subclasses. The number of possible inheritances from critical classes, by any rearrangement of the class hierarchy, is the product of the number of classes less one times the number of critical classes in the class hierarchy. In Figure 4.5, the classes which are shown to inherit from critical superclasses $A$ and $B$ are classes $B$, $E$, $F$, and $G$. If we apply the CSI metric to Figure 4.5, the value of this metric is calculated as follows. The number of classes which are in a position to inherit from each critical superclass is six, i.e. $5$ can inherit from class $A$ and $1$ can inherit from class $B$. The total number of possible inheritances from all critical classes in Figure 4.5 is twenty, i.e. for each of the 4 critical classes, there are 5 other classes that could potentially inherit from it, by a suitable rearrangement of the hierarchy.

Consider a set of classes in hierarchy $H$ as $C_i$, $i \in \{1, \ldots, c\}$, a set of critical classes in the same hierarchy as $CC_j$, $j \in \{1, \ldots, cc\}$, and a set of critical superclasses in the same hierarchy as $CSC_k$, $k \in \{1, \ldots, csc\}$ where $CSC \subseteq CC$ and $CC \subseteq C$. Let $\beta(CSC_k)$ be be the number of classes which may inherit from the critical superclass $CSC_k$ according to hierarchy $H$. Then, the Critical Superclasses Inheritance metric for hierarchy $H$ is defined as follows.

$$CSI(H) = \frac{\sum_{k=1}^{csc} \beta(CSC_k)}{(|C| - 1) \times |CC|} \quad (4.6)$$

Classified Methods Inheritance (CMI)  This metric is defined as “The ratio of the number of classified methods which can be inherited in a hierarchy to the total number of classified methods in that hierarchy”. It measures the proportion of classified methods which are exposed to inheritance by other classes since the presence of such methods increases “the size of the attack surface size” [40]. Hierarchies with lower values of CMI have fewer classified methods that are exposed to inheritance, and thus represent a more secure design. Figure 4.5 shows a classified method $N$ in class $B$ which is inheritable. The CMI metric measures this type of
method and penalises their use in a multi-class design. By contrast, Figure 4.5 has another inheritable classified method \( m \) in class \( I \). However, this method is not counted by this metric since the enclosing class \( I \) is not a superclass.

Consider a set of classified methods in hierarchy \( H \) as \( CM = \{cm_1, \ldots, cm_n\} \) and the classified methods which could be inherited in the same hierarchy as \( MI = \{mi_1, \ldots, mi_n\} \) such that \( MI \subseteq CM \). Then, we define the Classified Methods Inheritance metric as follows.

\[
CMI(H) = \frac{|MI|}{|CM|}
\]  

(4.7)

**Classified Attributes Inheritance (CAI)** We define this metric as “The ratio of the number of classified attributes which can be inherited in a hierarchy to the total number of classified attributes in that hierarchy”. It measures the proportion of classified attributes which are exposed to inheritance by other classes. Similar to the CMI metric, CAI aims to show that hierarchies with lower values of CAI have fewer classified attributes exposed to inheritance, and thus produce a more secure design with respect to the security design principle of “reducing the size of the attack surface” [40]. This metric can be illustrated using the classified attributes in Figure 4.5. Attribute \( \varepsilon \) in \( B \) is inheritable and is counted by this metric. However, attribute \( \zeta \) in class \( F \) is not inheritable in this design since class \( F \) is not a superclass. Therefore, it is not counted.

Consider a set of classified attributes in hierarchy \( H \) as \( CA = \{ca_1, \ldots, ca_n\} \) and the classified attributes which could be inherited in the same hierarchy as \( AI = \{ai_1, \ldots, ai_n\} \) such that \( AI \subseteq CA \). Then, we define the Classified Attributes Inheritance metric as follows.

\[
CAI(H) = \frac{|AI|}{|CA|}
\]  

(4.8)

### 4.2.5 Design Size

Design size measures the number of classes in a design [56]. A metric for measuring design size in object-oriented designs defined by Bansiya and Davis [56] is called Design Size in Classes (DSC). DSC is a count of the total number of classes in a certain design [56]. Bansiya and Davis’ study also revealed that design size has a major impact on a program’s reusability
Chapter 4. Security Metrics for Multi-Class Object-Oriented Designs

and functionality [56]. With respect to security, the size of object-oriented designs, to our
knowledge, has not been utilised. Nevertheless, Chowdhury et al.’s study [63] defines a metric
which measures the ratio of critical elements in a specific program’s code. This metric requires
the system to be fully implemented in order to calculate such a ratio.

Critical Design Proportion (CDP) This metric measures the impact of the size of a certain
design on security. We define it as “The ratio of number of critical classes to the total number
of classes in a design”. It measures the proportion of classes which store classified data, i.e.
critical classes. A higher proportion of critical classes in a design indicates higher security risks
for potential information flow. Designs with lower values of CDP indicate fewer critical classes
compared to other designs of the same size, and hence more secure systems with regard to the
security design principle of “reducing the attack surface size” [40].

Consider a set of classes in design \( D \) as \( C = \{c_1, \ldots, c_n\} \) and the critical classes in the
same design as \( CC = \{cc_1, \ldots, cc_n\} \) such that \( CC \subseteq C \). Then, we define the Critical Design
Proportion metric as follows.

\[
CDP(D) = \frac{|CC|}{|C|}
\]  

(4.9)

4.3 Multi-Class Design Metrics Case Study

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

4.3.1 Annotated UML Class Diagrams

This section shows an annotated class diagram for a planned computer program for the
Department of Defence. The class diagram in Figure 4.6 has been annotated using UMLsec
and SPARK’s annotations. The UMLsec annotations show the data which needs to be kept
confidential while SPARK’s annotations identify how data flows between the program’s classes,
attributes, and methods. The defence system class is responsible for storing information about
a person working within the Department of Defence. This person can be either an agent or an administrator. The Agent class is responsible for storing an agent’s information. This includes the agent’s name, job title, security clearance level and number. We assume that the security clearance level and number are codes which describe the security classification for a given agent. Details of an agent’s job title and security clearance are meant to be kept secret.

The Administrator class is responsible for storing information about an administrator. This information consists of the administrator’s name and job title. Unlike the Agent class, the Administrator class does not contain any confidential data. Both the Agent and Administrator classes use the same Name class.
4.3.2 Refactored UML Class Diagrams

In this section we illustrate the capabilities of our metrics by applying them first to the original design of the defence system, and then to three refactored versions of the original design using one or more of the refactoring rules defined by Fowler [18].

For instance, Figure 4.7 shows a design which has been constructed after applying a number of refactoring steps to the original one. It differs from the original design in the number of
classes. The system now has three classes instead of four, thanks to the refactoring rule *Inline Class* which has been applied to the original design through the merging of the *Clearance* and *Agent* classes. All the attributes and methods of the *Clearance* class have been moved to the *Agent* class by applying this rule.

Another refactored design is shown in Figure 4.8. It keeps the *Clearance* class but since the *Agent* and *Administrator* classes have similar features, it has used inheritance. To do this we applied the following refactoring rules: *Extract Superclass, Pull up Field*, and *Pull
up Method. Extract Superclass was used to combine similar attributes and methods in the Agent and Administrator classes by creating a superclass called Staff. The Pull up Field and Pull up Method refactoring rules were used to move the same fields and methods from both classes to the superclass. These fields are name and jobTitle, while the methods are GetName, SetJobTitle, and GetJobTitle. The same rules were also used to replace the constructors of both classes with one mutator called SetStaff. The only exception is keeping the Administrator job title field in its original class since it is not confidential and it can be exposed to the public.

Figure 4.9: Defence 4 Class Hierarchy

Figure 4.9 also used the Extract Superclass refactoring rule to separate a new class from the Administrator and Agent classes called Staff. Then it has used the Inline Class refactoring rule to combine the Clearance and Agent classes. It then used the Inline Class refactoring rule to combine the Staff and Name classes. Figure 4.9 has also labelled the critical class Agent, which contains classified attributes and methods, as ‘final’ to indicate that this class cannot be extended.
### 4.3.3 Security Metrics Results

Table 4.1 shows the results of applying our metrics to the four designs shown in Figures 4.6 to 4.9. For instance, to calculate the CPCC metric for Designs 1 and 2, we have to find out the number of critical composed-part classes and the total number of critical classes in both designs. There is one critical composed-part class in Design 1 while Design 2 has none. In addition, there are two critical classes in Design 1 while Design 2 has only one. According to the definition of the CPCC metric, we have to divide the number of composed-part critical classes by the total number of critical classes for each design to get its result.

The CCC metric counts the number of interactions with each classified attribute and divides it by the maximum number of possible interactions with these attributes. This number can be calculated by multiplying the number of the design’s classes less one by the number of classified attributes. In the case of Design 3, which has four classified attributes, the actual number of interactions with these attributes is two while the number of possible interactions is sixteen. Dividing these two numbers gives us 0.125 in this case. In the case of Design 1, which also has four classified attributes, the actual number of interaction with these attributes is also two but what differs from Design 3 is the the number of possible interactions which is twelve in this case.

The CCE metric is calculated by counting the number of extensible critical classes, which is zero in Design 4, and then dividing that by the total number of critical classes. By contrast, the remaining designs have not declared any of their critical classes as ‘final’ and hence their results with regard to this metric are one.
Additionally, the CME metric is calculated by counting the number of extensible classified methods, which is also zero in Design 4 and then diving that by the total number of classified methods. If the class is labelled as ‘final’ this means that it is not extensible and also means that all of its methods are final and not extensible even though they are not labelled as final. Since none of the remaining designs have neither declared any critical class as ‘final’ nor declared any classified methods as ‘final’, then their CME metric results have shown one. This means that all of their classified methods are extensible.

To calculate the CSP metric for Design 3, we first need the number of critical superclasses in all of the inheritance hierarchy, which is one. Then, this number is divided by the number of critical classes in the hierarchies, which is three. In the case of Design 4, which has only one critical subclass, the total number of critical superclasses is zero.

The CSI metric is calculated by first counting the number of classes which may inherit each critical superclass, which is two in Design 3. Then, we divide this number by the maximum number of classes which could possibly inherit from these critical superclasses, which is four in Design 3. Design 4 is another case which their exists inheritance hierarchy but it differs from Design 3 as the number of classes which may inherit each critical superclass is zero.

The CMI metric is calculated by counting the number of inheritable classified methods in all of the inheritance hierarchy, which is three in Design 3 divided by the number of classified methods in that hierarchy, which is five. However, Design 4 has used inheritance in a way that does not allow of its classified methods to be inherited, and therefore the CMI metric is zero in this regard.

The CAI metric is computed by dividing the number of inheritable classified attributes in all of the inheritance hierarchy, which is one in Design 3 by the number of classified attributes in that hierarchy, which is two. On contrast, Design 4 does not have any inheritable classified attributes, and hence the CAI metric is zero in this case.

Finally, Design 3 has the highest ratio of critical classes compared to other designs, so this design has the highest value of the CDP metric which is calculated by diving the number of critical classes in a design by the total number of classes in that design. On the other hand, Designs 2 and 4 have the lowest values of the CDP metric since they have only third of their total number of classes are critical.
4.4 Analysis of Security Metrics

An easy way of comparing the results of these metrics is to show them on radar charts (Figures 4.10 to 4.13). Given that our metrics are designed so that lower values are considered more secure, graphs which are closer to the centre of the charts indicate greater security. It can be seen that Design 4 shows the lowest values for all metrics compared to the other designs except for the CPCC metric, so we could say that this design has yielded the most secure design for all properties except composition. Designs which use composition in the same way have shown the same results. Designs 2 and 4 show the highest values for the CPCC metric. Therefore, these designs are the most insecure designs for composition. However, Design 1 is the most secure design with regard to the CPCC metric since it has the lowest ratio of critical composed-part classes.

With regard to the design property of coupling, Design 1 shows the least secure design since it has the highest value among the other designs for the CCC metric. This is because it has the highest number of links with classified attributes. By contrast, Design 2 and 4 are the most secure with regard to coupling. This is because Design 2 does not have any inter-class references or links with classified attributes while Design 4 has replaced the traditional association between classes with inheritance.
Chapter 4. Security Metrics for Multi-Class Object-Oriented Designs

Figure 4.11: Metrics for Defence 2

Figure 4.12: Metrics for Defence 3
With regard to extensibility, the only design which has declared all of its critical classes as ‘final’ is Design 4, which finalises all of its methods. This has resulted in this design being the most secure with regard to both the CCE and CME metrics. The other designs have yielded identical results since none have finalised any of their critical classes or classified methods.

With regard to inheritance, Design 4 has shown the lowest values for all of the inheritance metrics similarly to Designs 1 and 2 which have not used inheritance. Design 4 has used inheritance in a way such that there exists neither critical superclasses nor classified attributes or methods which could be inherited. Therefore, Designs 1, 2 and 4 are the most secure designs in this regard with a preference for Design 4 over the others since it has used inheritance in a secure way. On the other hand, Design 3 has shown the highest results, and hence the least secure design, in terms of all inheritance metrics.

With regard to the design size property, Designs 2 and 4 have the lowest ratio of critical classes compared to the total number of classes as the CDP metric shows. Thus, they are the most secure designs for this property. On the other hand, Design 3 is the least secure design for this property since its CDP metric reveals the highest ratio of critical classes in this design.

Clearly each design is good in some regards and poor in others. Another way of interpreting the results of our metrics is to choose which design satisfies the requirements of specific security
design principles of interest (i.e., “reducing the size of the attack surface” and “least privilege”). For instance, the metrics of CCC and CAI mostly contribute to the principle of “least privilege” and Designs 2 and 4 have yielded the lowest values with regard to these metrics, hence Design 2 and 4 are the most secure designs with regard to the principle of “least privilege”. On the other hand, from the requirements of the principle of “reducing the size of the attack surface”, Design 4 would be the most secure since the metrics of CPCC, CCE, CME, CSP, CMI, CAI and CDP have yielded the lowest values. In this case, we could conclude that Design 4 is the most secure design since it meets the requirements of both of the principles of “reducing the size of the attack surface” and “least privilege”. (In Chapter 7 we explore the problem of reconciling different metrics further by showing how composite metrics can be defined from sets of individual ones.)

4.5 Conclusion

In this chapter, we extended our previously defined design metrics which evaluate security based on a single class diagram to embrace entire class hierarchies. We defined several new security metrics for a complete object-oriented design. They can be used to compare different designs for the same program and identify the most secure one. These metrics have covered a number of software design quality properties consisting of: composition, coupling, extensibility, inheritance, and design size. They provide software designers with a simple approach for identifying where security vulnerabilities might occur from the perspective of information flow of confidential data, and thus with the ability to compare the security of various designs.

In the following chapter, we conduct a more general analysis to illustrate which refactoring methods can make certain classes more secure than others by making specific changes to the design.
Chapter 5

Security Assessment of Design-Level Refactoring Rules

Refactoring focuses on improving the reusability, maintainability and performance of programs, but the impact of refactoring on the security of a given program has received little attention. In this chapter we focus on the design of object-oriented applications and use our metrics from Chapters 3 and 4 to assess the impact of a number of standard refactoring rules on 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.

5.1 Introduction

Refactoring rules [18] 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. Using refactoring to enhance a program’s security has been considered in a few studies such as the work of Maruyama and Tokoda [100] who investigated how certain changes could affect the security characteristics of a given program with regard to access modifiers. Furthermore, Smith and Thober have identified a refactoring approach for critical systems [101]. Their approach aims to refactor a program’s code into two modules; a high-security and a low-security one. However, these previous approaches do not quantify the impact of changes
Chapter 5. Security Assessment of Design-Level Refactoring Rules

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.

To assess the impact of design refactoring rules on a program’s security, we start by first measuring the security of a given program’s design using our metrics from Chapters 3 and 4 and then measuring the security of other refactored designs of the same program. This allows us to identify those refactoring rules which can improve the security of a given object-oriented design.

5.2 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 our security design metrics from Chapters 3 and 4.

5.2.1 Identifying Security-Critical Design Refactoring Rules

Table 5.1 lists standard refactoring rules [18, 87] which are applicable at the design stage and which may affect the security level of an object-oriented design. We distinguish their specific effect on classified and non-classified features as we classify our data either as classified or non-classified as shown in Table 5.2 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’ [40] and ‘least privilege’ [33, 32]. The result of this assessment can inform programmers about the security significance of the refactoring rules which could help them while trying to maintain their code.

5.2.2 Assessing Security-Critical Design Refactoring Rules

In this section we analyse how the refactoring rules shown in Table 5.2 may affect our security design metrics defined in the previous two chapters.

For example, refactoring rules Encapsulate Classified Field and Hide Classified Method can improve security with regard to our Data Encapsulation-based metrics in two cases. One
Table 5.1: Design Refactoring Rules

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

is if non-private 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 non-private classified methods to make them private, this will reduce the COA metric, making the program more secure in this regard. These refactoring rules can also improve security with regard to the Inheritance-based metrics CMI and CAI if the affected attributes and methods are in an inheritance hierarchy and can be inherited (i.e., they are in critical superclasses).

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.
Chapter 5. Security Assessment of Design-Level Refactoring Rules

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.

<table>
<thead>
<tr>
<th>Security Refactoring Rule</th>
<th>Identifier</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encapsulate Classified Field</td>
<td>RNCF</td>
<td>Changes the access modifier of non-private classified fields to private.</td>
</tr>
<tr>
<td>Encapsulate Non-Classified Field</td>
<td>RNNF</td>
<td>Changes the access modifier of non-private non-classified fields to private.</td>
</tr>
<tr>
<td>Inline Classified Field</td>
<td>RICF</td>
<td>Combines two classified fields or more into one classified field if they are always used together.</td>
</tr>
<tr>
<td>Inline Non-Classified Field</td>
<td>RINF</td>
<td>Combines two classified and non-classified fields or more into one classified field if they are always used together.</td>
</tr>
<tr>
<td>Extract Classified Field</td>
<td>RECF</td>
<td>Creates a new classified field from an existing classified field if its information can be used separately.</td>
</tr>
<tr>
<td>Extract Non-Classified Field</td>
<td>RENF</td>
<td>Creates a new non-classified field from an existing classified or non-classified field if its information can be used separately.</td>
</tr>
<tr>
<td>Pull Up Classified Field</td>
<td>RPUCF</td>
<td>If two subclasses have the same classified field then this rule moves this field to their superclass.</td>
</tr>
<tr>
<td>Pull Up Non-Classified Field</td>
<td>RPUNF</td>
<td>If two subclasses have the same non-classified field then this rule moves this field to their superclass.</td>
</tr>
</tbody>
</table>

*Table 5.2: Security-Critical Design Refactoring Rules*
### Table 5.2 – Continued Security-Critical Design Refactoring Rules

<table>
<thead>
<tr>
<th>Security Refactoring Rule</th>
<th>Identifier</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push Down Classified Field</td>
<td>RPDCF</td>
<td>If a classified field is used by only some subclasses then this rule moves this field to those subclasses.</td>
</tr>
<tr>
<td>Push Down Non-Classified Field</td>
<td>RPDNF</td>
<td>If a non-classified field is used by only some subclasses then this rule moves this field to those subclasses.</td>
</tr>
<tr>
<td>Move Classified Field</td>
<td>RMCF</td>
<td>Moves a classified field to a critical class.</td>
</tr>
<tr>
<td>Move Non-Classified Field</td>
<td>RMNF</td>
<td>Moves a non-classified field to a critical class.</td>
</tr>
<tr>
<td>Hide Classified Method</td>
<td>RHCM</td>
<td>Makes non-private classified methods private if not used by another class.</td>
</tr>
<tr>
<td>Hide Non-Classified Method</td>
<td>RHNFM</td>
<td>Makes non-private non-classified methods private if not used by another class.</td>
</tr>
<tr>
<td>Inline Classified Method</td>
<td>RICM</td>
<td>Combines two classified methods or more into one classified method if they are always used together.</td>
</tr>
<tr>
<td>Inline Non-Classified Method</td>
<td>RINM</td>
<td>Combines two classified and non-classified methods or more into one classified method if they are always used together.</td>
</tr>
<tr>
<td>Extract Classified Method</td>
<td>RECM</td>
<td>Creates a new classified method from an existing classified method if its information can be used separately.</td>
</tr>
<tr>
<td>Extract Non-Classified Method</td>
<td>RENM</td>
<td>Creates a new non-classified method from an existing classified or non-classified method if its information can be used separately.</td>
</tr>
<tr>
<td>Finalise Classified Method</td>
<td>RFCM</td>
<td>Declares a classified method as “final” to prevent it from being extended.</td>
</tr>
</tbody>
</table>
Table 5.2 – Continued Security-Critical Design Refactoring Rules

<table>
<thead>
<tr>
<th>Security Refactoring Rule</th>
<th>Identifier</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finalise Non-Classified Method</td>
<td>RFNM</td>
<td>Declares a non-classified method as “final” to prevent it from being extended.</td>
</tr>
<tr>
<td>Pull Up Classified Method</td>
<td>RPUCM</td>
<td>If two subclasses have the same classified method then this rule moves this method to their superclass.</td>
</tr>
<tr>
<td>Pull Up Non-Classified Method</td>
<td>RPUNM</td>
<td>If two subclasses have the same non-classified method then this rule moves this method to their superclass.</td>
</tr>
<tr>
<td>Push Down Classified Method</td>
<td>RPDCM</td>
<td>If a classified method is used by only some subclasses then this rule moves this method to those subclasses.</td>
</tr>
<tr>
<td>Push Down Non-Classified Method</td>
<td>RPDNM</td>
<td>If a non-classified method is used by only some subclasses then this rule moves this method to those subclasses.</td>
</tr>
<tr>
<td>Move Classified Method</td>
<td>RMCM</td>
<td>Moves a classified method to a critical class.</td>
</tr>
<tr>
<td>Move Non-Classified Method</td>
<td>RMNM</td>
<td>Moves a non-classified method to a critical class.</td>
</tr>
<tr>
<td>Inline Critical Class</td>
<td>RICC</td>
<td>Combines two critical classes or more into one critical class if they are always used together.</td>
</tr>
<tr>
<td>Inline Non-Critical Class</td>
<td>RINC</td>
<td>Combines two critical and non-critical classes or more into one critical class if they are always used together.</td>
</tr>
<tr>
<td>Extract Critical Class</td>
<td>RECC</td>
<td>Creates a new critical class from an existing critical one.</td>
</tr>
<tr>
<td>Extract Composed-Part Critical Class</td>
<td>RECPCC</td>
<td>Creates a new composed-part critical class from an existing critical class.</td>
</tr>
</tbody>
</table>

Continued on next page
5.2. Assessment of Refactoring Rules

<table>
<thead>
<tr>
<th>Security Refactoring Rule</th>
<th>Identifier</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract Non-Critical Class</td>
<td>RENC</td>
<td>Creates a new non-critical class from an existing critical one.</td>
</tr>
<tr>
<td>Finalise Critical Class</td>
<td>RFCC</td>
<td>Declares a critical class as “final” to prevent it from being extended.</td>
</tr>
<tr>
<td>Finalise Non-Critical Class</td>
<td>RFNC</td>
<td>Declares a non-critical class as “final” to prevent it from being extended.</td>
</tr>
<tr>
<td>Extract Critical Super-class</td>
<td>RECSP</td>
<td>If two critical subclasses have similar classified features, this rule creates a critical superclass and moves these features into it.</td>
</tr>
<tr>
<td>Extract Non-Critical Superclass</td>
<td>RENSP</td>
<td>If two critical subclasses have similar non-classified features, this rule creates a non-critical superclass and moves these features into it.</td>
</tr>
<tr>
<td>Extract Critical Sub-class</td>
<td>RECSB</td>
<td>If two critical superclasses have similar classified features, this rule creates a critical subclass and moves these features into it.</td>
</tr>
<tr>
<td>Extract Non-Critical Subclass</td>
<td>RENSB</td>
<td>If two critical superclasses have similar non-classified features, this rule creates a non-critical subclass and moves these features into it.</td>
</tr>
</tbody>
</table>

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-extensible 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 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.

5.3 Design Refactoring Assessment Case Study

The following case study illustrates how applying the refactoring rules shown in Table 5.2 impacts the security of a design in a way measurable by our security design metrics, as predicted in Section 5.2. 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.

5.3.1 Original Annotated Design

The class diagram in Figure 5.1 has been annotated using UMLsec’s and SPARK’s annotations. The Bank Account system class hierarchy 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 CustomerAccount, a bank account can be either a savings or a credit account which is determined by the accountType attribute. The value of a Savings account’s
Figure 5.1: Bank Account Hierarchy 1
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 5.1 to be consistent with the UML class diagram rules. The CustomerAccount class also stores attributes of both the savings and credit accounts. The CreditCard 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.

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.

5.3.2 Refactored Annotated Designs

Figures 5.2 and 5.3 show two refactored versions of the original design using some of the refactoring rules in Table 5.2. 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 5.2 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 [56], 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-existing Address class to the Staff class. However, these steps make the program less secure with regard to its Design Size, and thus increase 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
Figure 5.2: Bank Account Hierarchy 2
5.3. Design Refactoring Assessment Case Study

Figure 5.3: Bank Account Hierarchy 3
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 areaCode and extensionNo, 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 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 5.3. It has been refactored from the original design in Figure 5.1 but it has used inheritance for the CustomerAccount and CreditCard classes for extensibility and effectiveness reasons [56]. 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 non-private classified instance and class attributes
and also non-private classified methods by using the *Encapsulate Classified Field* and *Hide Classified Method* rules. Such rules reduce the Data Encapsulation metrics CIDA, CCDA and COA. These rules can also reduce the Inheritance metrics CMI and CAI if the affected attributes and methods can be inherited, which is not so in this case. This has resulted in making the public classified instance and class attributes in all classes private. This has also changed the modifier access for two classified methods to be private (i.e., *VerifyCredit* in the *Credit* 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.

### 5.3.3 Security Design Metrics Results

Tables 5.3 and 5.4 show the results of applying our security design metrics from Chapters 3 and 4 to the three designs shown in Figures 5.1, 5.2 and 5.3.

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 5.2.2, 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 is the least secure design.

In comparison, Account 1 is more secure than Account 2 in most of the security-relevant metrics including the Data Encapsulation (CIDA and CCDA) Cohesion, Composition, Coupling, and Design Size-based metrics. Nevertheless, it shows the same results in terms of the Data Encapsulation (COA), Extensibility and Inheritance-based metrics, because none of the changes affect the accessibility of classified 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 measurably worse.

Furthermore, the security refactoring rules applied to Account 3 have improved the security level of other metrics including the Coupling, Extensibility 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, which made this design more secure in term of its Data Encapsulation metrics. 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.

5.3.4 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 or less secure as shown in Tables 5.5, 5.6 and 5.7. These lists show all of the security-critical refactoring rules in Table 5.2 and their expected impact on security for each of our security design metrics. Their impact is shown with regard to each security design metric in four different ways: “↑” means may increase that security metric, “↓” means may decrease that security metric, “↕” means may increase or decrease that security metric and “−” means has no impact on that metric. (Keep in mind that higher values of the metrics indicate worse security.) The final columns of the tables summarise how the refactoring rules affect the security of a design as 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. (We
### Table 5.5: Security-Critical Attributes Design Refactoring Rules Assessment

<table>
<thead>
<tr>
<th>Rule</th>
<th>CIDA</th>
<th>CCDA</th>
<th>COA</th>
<th>CMAI</th>
<th>CAAI</th>
<th>CAIW</th>
<th>CMW</th>
<th>CCC</th>
<th>CPCC</th>
<th>CCE</th>
<th>CME</th>
<th>CSP</th>
<th>CSI</th>
<th>CMI</th>
<th>CAI</th>
<th>CDP</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>RNCF</td>
<td>↓</td>
<td>↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>↓</td>
</tr>
<tr>
<td>RNNF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RICF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>↓</td>
</tr>
<tr>
<td>RINF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>↑</td>
</tr>
<tr>
<td>RECF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>↑</td>
</tr>
<tr>
<td>RENF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>↓</td>
</tr>
<tr>
<td>RPUCF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>↑</td>
</tr>
<tr>
<td>RPUNF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>↓</td>
</tr>
<tr>
<td>RPDCF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>↓</td>
</tr>
<tr>
<td>RPDNF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>↑</td>
</tr>
<tr>
<td>RMCF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>↓</td>
</tr>
<tr>
<td>RMNF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>↑</td>
</tr>
</tbody>
</table>
### Table 5.6: Security-Critical Methods Design Refactoring Rules Assessment

<table>
<thead>
<tr>
<th>Rule</th>
<th>RMNM</th>
<th>RMCM</th>
<th>RPDM</th>
<th>RPDCM</th>
<th>RPUM</th>
<th>RPUCM</th>
<th>RFNM</th>
<th>RFCM</th>
<th>RENM</th>
<th>RECM</th>
<th>RINM</th>
<th>RICM</th>
<th>RHN</th>
<th>RHCM</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIDA</td>
<td>↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCDA</td>
<td>↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COA</td>
<td>↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMAI</td>
<td>↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAAI</td>
<td>↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAGW</td>
<td>↑</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMW</td>
<td>↑</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCCC</td>
<td>↑</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPGC</td>
<td>↑</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCE</td>
<td>↑</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMME</td>
<td>↑</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CSP</td>
<td>↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CSI</td>
<td>↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMI</td>
<td>↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAI</td>
<td>↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CDP</td>
<td>↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>↑</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule</td>
<td>RICC</td>
<td>RINC</td>
<td>RECC</td>
<td>RENC</td>
<td>RFCC</td>
<td>RFNC</td>
<td>RECPCC</td>
<td>RENSP</td>
<td>RECSP</td>
<td>RECSB</td>
<td>RENSB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CIDA</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCDA</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COA</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMAI</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAI</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMI</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CSI</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CSP</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CME</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCE</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPCC</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCC</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMW</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMI</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCAI</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAIW</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
discuss this further in Chapter 7. Tables 5.5, 5.6, and 5.7 show 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.

## 5.4 Conclusion

In this chapter, we have assessed how refactoring steps can influence the security of a given object-oriented design as measured by a set of security metrics. These refactoring rules may have an impact on the size of the design’s ‘attack surface’ and whether or not it grants the ‘least privilege’ over classified data. The results of this assessment show that specific refactoring rules can improve the overall security of a given program from the point of view of potential information flow in a quantifiable way when used to restrict the accessibility of classified data. This therefore validates the effectiveness of our security metrics and confirms our expectation for the refactoring rules. In particular, our assessment has shown that there are twenty different common refactoring rules which can improve the overall security of a given design, twelve rules which make security worse and the remaining four rules have no impact on security overall. This assessment can serve to guide the application of refactoring rules to security-critical programs, alerting application designers to potentially unsafe refactoring steps at an early stage.

This study allows us to identify those refactoring rules which improve our security metrics because they implement general strategies for mitigating API-level security vulnerabilities. For instance, the refactoring rule of Encapsulate Classified Fields allows us to make public classified fields private, and hence mitigate the API-level security vulnerability of non-authorised objects directly reading security-critical attributes. Applying this rule improves the CIDA and CCDA security metrics. Another example is the refactoring rule of Hide Classified Methods which allows us to make public methods that access classified attributes private, and hence mitigate the security vulnerabilities of non-authorised objects accessing classified information through public methods which in turn improves the COA security metric. Furthermore, we can avoid the threat of non-authorised objects extending critical classes or classified methods to maliciously override them by using those refactoring rules which prevent this activity, i.e., Finalise Critical Class and Finalise Classified Method. Applying these rules improves the CCE and CME security metrics.
The metrics can be calculated automatically by a UML tool once the design is augmented with UMLsec’s annotations (which identify the confidential data) and SPARK’s annotations (which show how data flows within the program). We have implemented such a tool and describe it in Chapter 10.
Part III

OBJECT-ORIENTED CODE-LEVEL SECURITY ASSESSMENT
Chapter 6

Security Metrics for Object-Oriented Programs

In this chapter we extend our design-level security metrics to produce metrics for object-oriented programs which are capable of assessing the security of a Java program from the point of view of potential information flow. These metrics can be used to compare the security of programs or assess the effect of program modifications on security. We have developed a tool which automatically measures the security of a given Java bytecode program in terms of the accessibility of distinguished ‘classified’ attributes (Chapter 10). We apply our tool to a number of open-source Java programs to show how these new security metrics work.

6.1 Introduction

There exist various approaches which aim to reduce security risks and vulnerabilities in software either through careful coding practices [3, 2] or through static analysis of the code’s properties [9, 116, 117, 118, 119, 120]. However, these techniques require considerable skill and effort to apply successfully, and are not always applicable to pre-existing programs. Alternatively, security metrics which quantify the security level of a given program [63] offer a ‘pushbutton’ solution which can be applied easily to given programs.

Dynamic metrics have been studied in many projects due to their importance and reliability [121], including the relationships between static and dynamic coupling metrics [122]. Java dynamic metrics have also been studied extensively, including the work of Dufour et
al. [121] who define a number of dynamic metrics for Java in order to evaluate compiler optimisations. Another relevant study is that of Binder and Hulass into control flow metrics [123]. However, none of these existing projects have developed metrics for program security.

In this chapter, we extend our work in Chapters [3] and [4] which defined several security metrics for UML designs in order to define a number of additional security metrics for Java programs. To make it easy to use the security metrics, we have developed a tool which we call the Java Bytecode Security Analyser (JBSA) described in Chapter [10]. To use it the programmer first defines which program variables contain “classified” data and the tool then automatically evaluates the bytecode’s security with respect to the accessibility of this data. This chapter defines the metrics implemented by the tool.

6.2 Program Code Security Metrics Definitions

This section defines our security metrics for assessing the security level of programs. Our approach is based on static analysis of the program code. For instance, when we say that a method ‘accesses’ or ‘interacts with’ a classified attribute, it means that the method’s compiled code contains an instruction that may read or write an attribute labelled by the programmer as ‘classified’. Of course, if this instruction appears within a conditional statement, there is no guarantee that the method will do so every time the program executes. Thus, our metrics are safely conservative and measure the potential flow of classified data.

6.2.1 Data Encapsulation-Based Security Metrics

Code security metrics based on data encapsulation aim to statically measure the potential flow of information from classified attributes and methods from the perspective of access modifiers. Specific Java access modifiers include: public, protected, package-private and private. These metrics consider attributes annotated by the programmer as ‘classified’ or ones which derive their values from classified attributes, and methods which access classified attributes. They also measure whether the overall program imports the Java reflection library due to the risk that this library can be used by new code to access the value of any classified attribute in an existing security-critical part of the program [124].
We divide the metrics for this property into four kinds of accessibility: classified instance attributes (CIDA); classified class attributes (CCDA); methods which access classified attributes (COA) and accessibility through reflection (RPB). The metrics are defined so as to penalise programs that make classified attributes more accessible. Higher values indicate higher accessibility to classified attributes and methods, and hence a larger ‘attack surface’. This means a higher possibility for confidential data to be exposed to unauthorised parties. Aiming for lower values of these metrics adheres to the security principle of reducing the size of the attack surface \[40, 124\]. Here we define the security metric RPB while the descriptions and definitions of the CIDA, CCDA and COA metrics are shown in Chapter \[3\].

**Reflection Package Boolean (RPB)** This code-level metric measures the accessibility through reflection of classified data in a given program. It is defined as “A boolean value representing whether the Java program imports the Java reflection package (1) or not (0)”. This metric is only concerned with whether or not the application itself is importing the Java reflection library (i.e., information flow within the program itself) and does not consider an attacker reflecting on the application code elsewhere. The reflection metric equals 1 if the program imports the Java reflection library or 0 otherwise. Importing the Java reflection library means a higher possibility for confidential data to be exposed to unauthorised parties.

\[
R_{PB}(P) = \begin{cases} 
1, & \text{if reflection imported} \\
0, & \text{otherwise} 
\end{cases}
\quad (6.1)
\]

### 6.2.2 Cohesion-Based Security Metrics

The property of cohesion measures the interactions between attributes and methods within a given class \[53\]. In Chapter \[3\], we defined four cohesion-based security metrics to measure the potential flow of classified information caused by interactions between methods and classified attributes in an object-oriented design. Programs with higher interaction between methods and classified attributes have stronger cohesion, and hence are less secure. The previously-defined metrics are divided into four parts: the interactions of mutators (setters) with classified attributes (CMAI); the interactions of accessors (getters) with classified attributes (CAAI); the weight of classified attributes’ interactions with methods (CAIW); and the proportion of classified
methods (CMW). In this chapter, we extend these design-level metrics to include the proportion of classified writing methods in the program (CWMP).

**Classified Writing Methods Proportion (CWMP)** This metric aims to measure the proportion of methods which write classified attributes in a particular program. We define this metric as: "The ratio of the number of methods which write classified attributes to the total number of classified methods in the program". For our purposes, we assume a writing method in Java is one which writes an attribute to outside its class by calling a method from the `java.io` package. This includes methods whose class name contains either ‘write’, ‘print’, or ‘out’. Therefore, a ‘classified writer’ is a method which uses one of these classes to write a classified attribute. Fewer such methods adheres to the principle of granting least privilege [32]. (Of course, in practice this process cannot measure input and output that occurs in external libraries that are not available to our analysis tool. See Chapter 10.)

Consider the set of classified methods in a program $P$ as $CM = \{cm_1, \ldots, cm_n\}$ and the set of the classified writing methods as $CWM = \{cwm_1, \ldots, cwm_n\}$ such that $CWM \subseteq CM$. (Given a set $S$, let the magnitude operator $|S|$ returns the size of the set.) Then, $CWMP$ is expressed as:

$$CWMP(P) = \frac{|CWM|}{|CM|} \quad (6.2)$$

### 6.2.3 Coupling-Based Security Metric

Our security coupling metric (CCC) from Chapter 4 measures the degree of potential flow of classified data caused by the interactions between classes and classified attributes in a given object-oriented design. This metric is adopted without change for program code.

### 6.2.4 Composition-Based Security Metric

As explained in Chapter 4, composition yields a (weak) possibility of potential information flow for classified data. In the case of programming in Java, it is possible to access composed-part (inner) classes unless they are marked as private. As a result it is recommended to avoid using non-private inner classes in security-critical code [13]. Thus, in our case we assume that
using private composed-part classes should reduce the potential for flow of classified data, and hence produce more secure programs. Our design-level composition-based metric (CPCC) from Chapter 4 is adopted for program code to measure this.

### 6.2.5 Extensibility-Based Security Metrics

To have more secure programs, classes and methods which can access classified data should be prevented from being extended by other classes and methods [13, 115], since doing so makes classified data accessible in the new code. We have identified two metrics in Chapter 4 (CCE and CME) which reward the use of non-extended classes and methods.

Another such threat with regard to this property is code that assigns classified values to a variable or parameter that is not subsequently used, because this makes it possible to add code to the program that accesses the classified data but has no observable effect on the program’s behaviour. To prevent this we need to identify classified values that are defined but not used, and classified methods that are declared but not called. Thus, we define three new code-level metrics which penalise unused classified attributes (UACA), uncalled classified methods (UCAM) and unused critical classes (UCAC) in a program.

These features could allow unauthorised parties to acquire privileges on security-critical data without affecting the program’s original behaviour. Making code inextensible eliminates this risk, and hence reduces the possibility of information flow from these attributes, methods and classes, which adheres to the principle of granting the least privilege [32].

**Unaccessed Assigned Classified Attribute (UACA)** This metric is defined as “The ratio of the number of classified attributes that are assigned but never used to the total number of classified attributes in the program”. It measures those classified attributes which are assigned, either directly by an “=” assignment or by parameter passing through value or reference, but never subsequently used.

Consider the set of classified attributes in a program $P$ as $CA = \{ca_1, \ldots, ca_n\}$ and the set of classified attributes which are assigned but never used in the same program as $UCA = \{uca_1, \ldots, uca_n\}$, such that $UCA \subseteq CA$. Then, we define the Unaccessed Assigned Classified Attribute metric as follows.
Chapter 6. Security Metrics for Object-Oriented Programs

\[ UACA(P) = \frac{|UCA|}{|CA|} \]  

Uncalled Classified Accessor Method (UCAM) This metric measures declared methods which access classified attributes but are never called. It is defined as “The ratio of the number of classified methods that access a classified attribute but are never called by other methods to the total number of classified methods in the program”.

Consider a set of classified methods in a program \( P \) as \( CM = \{cm_1, \ldots, cm_n\} \) and classified accessors that are never called by other methods \( UCM = \{ucm_1, \ldots, ucm_n\} \), such that \( UCM \subseteq CM \). Then, we define the Uncalled Classified Accessor Method metric as follows.

\[ UCAM(P) = \frac{|UCM|}{|CM|} \]

Unused Critical Accessor Class (UCAC) This measures classes which contain classified accessor methods that are never used in any other classes. It is defined as “The ratio of the number of classes which contain classified methods that access classified attributes but are never used by other classes to the total number of critical classes in the program”.

Consider the set of critical classes in a program \( P \) as \( CC = \{cc_1, \ldots, cc_n\} \) and classes which have classified accessors that are never used by other classes as \( UCC = \{ucc_1, \ldots, ucc_n\} \), such that \( UCC \subseteq CC \). hen, we define the Unused Critical Accessor Class metric as follows.

\[ UCAC(P) = \frac{|UCC|}{|CC|} \]

6.2.6 Design Size-Based Security Metrics

With respect to security, a program with a large amount of security-critical code has a higher chance of potential flow of classified information, and hence is less secure \([63]\). It has also been shown that security-sensitive classes must avoid serialisation since this allows the values of private fields to be accessed from outside the program \([124]\). Our design size-based security
metric (CDP) defined in Chapter 4 already measures the proportion of the program that is devoted to security-critical classes (CDP).

For program code we define another security metric devoted to security-critical serialisable classes (CSCP).

**Critical Serialised Classes Proportion (CSCP)** This metric measures the risk associated with critical serialisable classes in a given program. We define it as “The ratio of the number of critical serialised classes to the total number of critical classes in the program”. It rewards programs with a smaller percentage and number of such classes and penalises the use of security-critical serialisable classes. Therefore, lower values of the CSCP metric indicates a lower proportion of security-critical serialisable classes, which can give privileges over confidential data, and thus satisfies the least privilege principle [32].

Consider the set of the critical classes in program P as \( CC = \{cc_1, \ldots, cc_n\} \) and the set of critical serialised classes is \( CSC = \{csc_1, \ldots, csc_n\} \), such that \( CSC \subseteq CC \). Then, we define the Critical Serialised Classes Proportion metric as follows.

\[
CSCP(P) = \frac{|CSC|}{|CC|}
\]

6.2.7 Inheritance-Based Security Metrics

The design-level metrics from Chapter 4 which consider inheritance are equally-applicable to program code. The include metrics which penalise classes (CSP and CSI), methods (CMI), and attribute (CAI) hierarchies in which classified data appears near the top.

6.3 Program Code Security Metrics Experimental Results

To demonstrate the validity of our code-level metrics, and test our Java Bytecode Security Analyser described in Chapter 10 we conducted an experiment with several large-scale open source Java programs. We used the tool to assess the relative security of different versions of the same program. Our hypothesis was that a program’s level of security should, on average, improve over time, as bugs are fixed and the program code is improved, although the addition
of new security-critical code may cause a worsening of overall security levels.

6.3.1 Approach

We began with five existing open source Java security projects. (Although we considered using open source software from the Java Qualitas Corpus [125], most of the software in this source either does not have multiple versions or is not security related. Instead, our projects were chosen from the most frequently downloaded security projects on the SourceForge website [126].) For each project, we chose a specific version which was modified in a number of subsequent updates, to fix bugs found in the previous releases. In this way we could compare different versions of each program with identical functionality but (hopefully) improved code quality. All of these programs are security-related, so we could reasonably expect successive releases to be more secure than their predecessors.

6.3.2 Programs Analysed

The chosen programs consisted of the following: Jacksum, jGuard, Kasai, JSecurity, JXplorer. JSecurity provides a framework for handling authentication, authorization, enterprise session management, and cryptography services [126]. Kasai is an authentication and authorisation framework which provides the integration of a simple manageable permission scheme into applications [126]. Jacksum is a Java program which is designed to compute and verify checksums, CRCs and hashes [126]. jGuard is a program for web and standalone applications responsible for providing a security framework based on Java authentication and authorisation security to resolve access control problems [126]. Finally, JXplorer provides a Java ldap browser with the support of LDIF, SSL, SASL and GSSAPI security schemes [126].

6.3.3 Program Annotations

A challenge we faced is that the programs need to be annotated, to identify classified data items, before we can analyse them. Therefore, we manually annotated at the Java source code level a number of attributes in each project to be ‘classified’, choosing attributes whose names and associated code comments indicated that they are likely to store confidential data. We annotated the same attributes for all the different releases of the same program in
order to make our comparisons fair. For example, in the program JSecurity, we annotated
the following attributes as classified: username, password and rememberme in the UsernamePasswordToken class.

With regard to the Kasai project, we annotated five different attributes as classified: login, password and superUser in the User class and in class Role the classified attributes were id and name.

In the Checksum project, we annotated the following attributes to be classified: value, length, separator, and filename in class AbstractChecksum. Additionally, we annotated attribute val in class Crc16 to be classified.

With regard to the JGuard project, we annotated four attributes as classified: name and applicationName in the JGuardPrincipal class and id and value in the JGuard-Credential class.

In the JXplorer project, we chose the following attributes to be treated as classified when we were analysing the security of this project: uniqueID and addressIP in the Name class and tag and name in the ASN1Type class.

6.3.4 Program Characteristics

Table 6.1 shows a number of static characteristics of the studied programs after we annotated our choices of classified attributes for each. The arrows show how each metric has changes since the previous release. Upwards arrows (red) indicate a worsening of security and downwards arrows (green) indicate that security has improved. (These characteristics are one of the outputs of the JBSA tool.) They include the total number of attributes, classified attributes, methods, classified methods, classes and critical classes for each program. In each successive release of each project, most of these characteristics either grew or stayed the same. For instance, the number of classified methods i.e., those which may access our annotated attributes, either directly or indirectly, can be seen to grow dramatically in successive revisions of Jacksum and JXplorer.

In order to show that our security metrics reflect the program’s true security level, we inspected the code of some of the analysed programs in this experiment. This inspection aimed to show that our security metrics correctly mirror the improvement or worsening of security
caused by specific changes to security-relevant code.

```
public void overridePassword(String newPassword) throws ServiceNotAvailableException, ServiceException, InvalidAttributesException, InvalidPasswordException {
    ResourceBundle res = ResourceBundle.getBundle(Constants.PROPERTY_FILE);

    AuthService authService = AuthServiceFactory.getAuthService(res.getString("auth.service"));

    authService.setPassword(this.login, StringUtls.defaultString(newPassword));
    this.setPassword(newPassword);
}
```

```
public void setPassword(String password) {
    this.password = password;
}
```

**Figure 6.1**: Setting Password Methods in Kasai 1.1.0.B2

For instance, it can be seen in program Kasai that the second release has added a number
of additional methods some of which contain a flow of classified information. One such new method is `overridePassword` (shown in Figure 6.1) in class `User` which interacts with the classified attribute `password` and does similar operations to another existing method `setPassword`. It thus creates an additional access point for classified attributes.

Similarly the second release of the Kasai program overloads an existing security-critical method. Class `KasaiFacade` has two methods called `createUser` that have a flow of classified information as shown in Figure 6.2.

```
public void createUser(String loginUser, String idUser, String firstName, String lastName,
                       String email, boolean blocked, String description, boolean superUser, String clientIP) 
    throws DataAccessException, AlreadyExistsException, InvalidAttributesException,
           DoesntExistException, NotEnoughPermissionException, CannotAuditException,
           CriticalException {

    Log.getInstance(Version.PROPERTY_FILE).write(KasaiFacade.class.getName(), "createUser", "Enter", java.util.logging.Level.INFO);

    long startTime = System.currentTimeMillis();
    String raisedError = null;
    int returnCode = 0;

    try {
        this.validateOperative(loginUser, KasaiFacade.COMMIT_USER, "/kasai/user/");

        boolean sU = this.readUser(loginUser).getSuperUser();
    }

    public void createUser(String loginUser, String idUser, String firstName, String lastName,
                       String email, boolean blocked, String description, boolean superUser, String password,
                       String clientIP) 
    throws DataAccessException, AlreadyExistsException, InvalidAttributesException,
           DoesntExistException, NotEnoughPermissionException, CannotAuditException,
           InvalidPasswordException {

    Log.getInstance(Version.PROPERTY_FILE).write(KasaiFacade.class.getName(), "createUser", "Enter", java.util.logging.Level.INFO);

    long startTime = System.currentTimeMillis();
    String raisedError = null;
    int returnCode = 0;

    try {
        this.validateOperative(loginUser, KasaiFacade.COMMIT_USER, "/kasai/user/");

        boolean sU = this.readUser(loginUser).getSuperUser();
    }
```

Figure 6.2: Two Overloaded Methods in Kasai 1.1.0.B2

This means that there are more methods in this release which interact with classified information than in the previous release. In fact, these new methods have similar responsibilities as existing ones and could have been avoided.
Another example can be seen in program Jacksum 1.3 which has a method getChecksumInstance that returns classified information and is assigned to a new non-classified attribute checksum as shown in Figure 6.3. The method calling getChecksumInstance thus exposes classified information which could be exploited by unauthorised parties. Therefore, we expect the security of Jacksum 1.3 to worsen due to the additional vulnerabilities added to it and our security metrics should reflect this change.

![Figure 6.3: Jacksum 1.3 Potential Vulnerability](image)

On the other hand, there are cases where a potential vulnerability has been removed from the program in a successive release. This can be seen in program JSecurity where method executeLogin in class FormAuthenticationFilter that used to be a potential vulnerability in the third release (shown in Figure 6.4) was deleted from the program’s fourth release. This has resulted in reducing the number of insecure methods (i.e., those which interact with security-critical information). Such changes could contribute to improvements in the program’s overall security and therefore our security metrics should reflect this improvement.

### 6.3.5 Programs Security Metrics Results

The results of calculating our code-level security metrics (using our Java Bytecode Security Analyser) for each release of each project are summarised in Tables 6.2 to 6.4. Given that lower values of each metric are considered more secure, programs whose metrics decrease should be those whose security has improved. We expected that these security-related programs would improve their overall security with each new release.

With regard to the results shown in the tables, two of the five programs, JSecurity and Jacksum, show an obvious improvement in their security metrics from previous versions. One exception is jGuard whose metrics are unchanged for all releases. This suggests that only
### Table 6.2: Data Encapsulation and Cohesion-Based Security Metrics

<table>
<thead>
<tr>
<th>Program</th>
<th>Version</th>
<th>CIDA</th>
<th>CCDA</th>
<th>COA</th>
<th>RPB</th>
<th>CMAI</th>
<th>CAAI</th>
<th>CAIW</th>
<th>CMW</th>
<th>CWMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSecurity</td>
<td>0.9.0.A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0.0228</td>
<td>0.0037</td>
<td>0.0225</td>
<td>0.0134</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0.9.0.B1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0.0226</td>
<td>0.0036</td>
<td>0.0223</td>
<td>0.0132</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0.9.0.RC1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0.0208</td>
<td>0.0032</td>
<td>0.0205</td>
<td>0.012</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0.9.Stable</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0.0207</td>
<td>0.0032</td>
<td>0.0206</td>
<td>0.0113</td>
<td>0</td>
</tr>
<tr>
<td>Kasai</td>
<td>1.1.0.B1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.031</td>
<td>0.028</td>
<td>0.115</td>
<td>0.09</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1.1.0.B2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.033</td>
<td>0.03</td>
<td>0.125</td>
<td>0.11</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1.1.0.B3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.033</td>
<td>0.03</td>
<td>0.125</td>
<td>0.11</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1.1.Stable</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.033</td>
<td>0.03</td>
<td>0.125</td>
<td>0.11</td>
<td>0</td>
</tr>
<tr>
<td>Jacksum</td>
<td>1.2</td>
<td>-0.8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.05</td>
<td>0.034</td>
<td>0.08</td>
<td>0.156</td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td>-0.6</td>
<td>0</td>
<td>0.976</td>
<td>0</td>
<td>0</td>
<td>0.047</td>
<td>0.044</td>
<td>0.116</td>
<td>0.189</td>
</tr>
<tr>
<td></td>
<td>1.4</td>
<td>-0.4</td>
<td>0</td>
<td>0.931</td>
<td>0</td>
<td>0</td>
<td>0.029</td>
<td>0.034</td>
<td>0.088</td>
<td>0.145</td>
</tr>
<tr>
<td></td>
<td>1.5</td>
<td>-0.25</td>
<td>0</td>
<td>0.928</td>
<td>0</td>
<td>0</td>
<td>0.022</td>
<td>0.03</td>
<td>0.123</td>
<td>0.158</td>
</tr>
<tr>
<td>jGuard</td>
<td>0.65.1</td>
<td>0.65</td>
<td>0</td>
<td>0.85</td>
<td>0</td>
<td>0</td>
<td>0.026</td>
<td>0.026</td>
<td>0.036</td>
<td>0.116</td>
</tr>
<tr>
<td></td>
<td>0.65.2</td>
<td>0.65</td>
<td>0</td>
<td>0.85</td>
<td>0</td>
<td>0</td>
<td>0.026</td>
<td>0.026</td>
<td>0.036</td>
<td>0.116</td>
</tr>
<tr>
<td></td>
<td>0.65.3</td>
<td>0.65</td>
<td>0</td>
<td>0.85</td>
<td>0</td>
<td>0</td>
<td>0.026</td>
<td>0.026</td>
<td>0.036</td>
<td>0.116</td>
</tr>
<tr>
<td></td>
<td>0.65.4</td>
<td>0.65</td>
<td>0</td>
<td>0.85</td>
<td>0</td>
<td>0</td>
<td>0.026</td>
<td>0.026</td>
<td>0.036</td>
<td>0.116</td>
</tr>
<tr>
<td>JXplorer</td>
<td>3.2.B1</td>
<td>0.8</td>
<td>0.01</td>
<td>0.94</td>
<td>1</td>
<td>0.0025</td>
<td>0.0028</td>
<td>0.081</td>
<td>0.131</td>
<td>0.007</td>
</tr>
<tr>
<td></td>
<td>3.2.B2</td>
<td>0.8</td>
<td>0.008</td>
<td>0.92</td>
<td>1</td>
<td>0.0025</td>
<td>0.0026</td>
<td>0.088</td>
<td>0.146</td>
<td>0.006</td>
</tr>
<tr>
<td></td>
<td>3.2.B3</td>
<td>0.81</td>
<td>0.02</td>
<td>0.92</td>
<td>1</td>
<td>0.0025</td>
<td>0.0024</td>
<td>0.105</td>
<td>0.167</td>
<td>0.007</td>
</tr>
<tr>
<td></td>
<td>3.2.Stable</td>
<td>0.83</td>
<td>0.019</td>
<td>0.92</td>
<td>1</td>
<td>0.0025</td>
<td>0.0024</td>
<td>0.109</td>
<td>0.155</td>
<td>0.008</td>
</tr>
</tbody>
</table>

### Table 6.3: Coupling, Composition and Extensibility-Based Security Metrics

<table>
<thead>
<tr>
<th>Program</th>
<th>Version</th>
<th>CCC</th>
<th>CPCC</th>
<th>CCE</th>
<th>CME</th>
<th>UACA</th>
<th>UCAM</th>
<th>UCAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSecurity</td>
<td>0.9.0.A</td>
<td>0.0111</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.75</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0.9.0.B1</td>
<td>0.0109</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.75</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0.9.0.RC1</td>
<td>0.0103</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.75</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0.9.Stable</td>
<td>0.0054</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.75</td>
<td>0</td>
</tr>
<tr>
<td>Kasai</td>
<td>1.1.0.B1</td>
<td>0.036</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.38</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1.1.0.B2</td>
<td>0.04</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.36</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1.1.0.B3</td>
<td>0.04</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.36</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1.1.Stable</td>
<td>0.04</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.36</td>
<td>0</td>
</tr>
<tr>
<td>Jacksum</td>
<td>1.2</td>
<td>0.087</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.71</td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td>0.061</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td>1.4</td>
<td>0.048</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td>1.5</td>
<td>0.033</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.85</td>
</tr>
<tr>
<td>jGuard</td>
<td>0.65.1</td>
<td>0.0625</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>0.65.2</td>
<td>0.0625</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>0.65.3</td>
<td>0.0625</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>0.65.4</td>
<td>0.0625</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.6</td>
</tr>
<tr>
<td>JXplorer</td>
<td>3.2.B1</td>
<td>0.0098</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.029</td>
</tr>
<tr>
<td></td>
<td>3.2.B2</td>
<td>0.0094</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.032</td>
</tr>
<tr>
<td></td>
<td>3.2.B3</td>
<td>0.0086</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>3.2.Stable</td>
<td>0.0082</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.037</td>
</tr>
</tbody>
</table>
insignificant changes were made to the code which had no security impact at all. Indeed, this impression is confirmed by the characteristics in Table 6.1 which reveal that no major changes were made to the code’s size between releases. (Nevertheless, the program’s change log says that a number of small bug fixes were made in each revision.)

The other exception is Kasai whose metrics in Table 6.2 have slightly increased in value between releases 1.1.0.B1 and 1.1.0.B2, meaning a worsening in security, after which the program was stable. From Table 6.2 this would appear to be because the second release added eight new methods, five of which were ‘classified’. These new methods account for the slight increase in three of the cohesion and coupling-security related metrics exhibited by the second version of the program.

A similar case is shown by the results of JXplorer where some of its metrics often increased. The reason for this is clearly shown by the program characteristics in Table 6.1 which indicate that the program has had a significant amount of new code added. Thus, the program has major increases in some of its security metrics and worse security overall with regard to those metrics. Nevertheless, some of the program’s metrics, including COA, CAAI, CCC and CSCP, have managed to decrease and thus its security has improved in these particular respects.

Comparing these results with the code inspections described in Section 6.3.4 we see that our security metrics for these programs have accurately reflected the changes in the security of these programs with regard to either removing or adding potential security vulnerabilities.
Table 6.4: Design Size and Inheritance-Based Security Metrics

<table>
<thead>
<tr>
<th>Program</th>
<th>Version</th>
<th>CDP</th>
<th>CSCP</th>
<th>CSP</th>
<th>CSI</th>
<th>CMI</th>
<th>CAI</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSecurity</td>
<td>0.9.0.A</td>
<td>0.0041</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0.9.0.B1</td>
<td>0.0041</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0.9.0.RC1</td>
<td>0.0038</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0.9.Stable</td>
<td>0.0032</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Kasai</td>
<td>1.1.0.B1</td>
<td>0.039</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1.1.0.B2</td>
<td>0.039</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1.1.0.B3</td>
<td>0.039</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1.1.Stable</td>
<td>0.039</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Jacksum</td>
<td>1.2</td>
<td>0.083</td>
<td>0</td>
<td>0.5</td>
<td>0.174</td>
<td>0.722</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td>0.103</td>
<td>0</td>
<td>0.5</td>
<td>0.232</td>
<td>0.722</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td>1.4</td>
<td>0.083</td>
<td>0</td>
<td>0.5</td>
<td>0.2</td>
<td>0.722</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td>1.5</td>
<td>0.068</td>
<td>0</td>
<td>0.5</td>
<td>0.221</td>
<td>0.75</td>
<td>0.8</td>
</tr>
<tr>
<td>jGuard</td>
<td>0.65.1</td>
<td>0.044</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0.65.2</td>
<td>0.044</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0.65.3</td>
<td>0.044</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0.65.4</td>
<td>0.044</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>JXplorer</td>
<td>3.2.B1</td>
<td>0.069</td>
<td>0.036</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3.2.B2</td>
<td>0.068</td>
<td>0.036</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3.2.B3</td>
<td>0.075</td>
<td>0.032</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3.2.Stable</td>
<td>0.074</td>
<td>0.032</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

For instance, in Kasai’s second release our security metrics that directly relate to measuring the security of classified methods have shown that security has worsened in this release as expected due to the addition of new security-critical methods and attributes. This is also the case for program Jacksum 1.3. On the other hand, security improved for the JSecurity program’s fourth release as a potentially vulnerable method was deleted. This change in the code produced a corresponding decrease in metrics that measure the proportion of classified methods (CMAI and CMW). However because the deleted classified methods also interacted with several non-classified attributes, the metric that measures the proportion of classified interactions (CAIW) increased.

From this experiment, we can conclude that our security metrics offer a simple, easy to apply and easy to interpret approach to quantifying the security of a given program, once it is properly annotated. In Chapter [7], we also explain how these sets of metrics can be meaningfully summarised to produce a smaller set of numerical results, to save the programmer the need to interpret large tables of figures. The tool described in Chapter [10] can produce this abbreviated data too.
6.4 Conclusion

In this chapter we have defined a number of security metrics for object-oriented programs. The metrics provide software developers with a simple way of identifying and fixing security vulnerabilities which might occur from the perspective of information flow of confidential data. The security metrics were demonstrated using actual large-scale Java projects. This case study produced results which matched our findings from inspecting the code about the way the programs’ security changed as their code was extended or debugged.

In the next chapter, we use these metrics to present a new hierarchical model for assessing the overall security of a given object-oriented program based on a static analysis of its code. The model consists of four hierarchical levels and provides an easy-to-interpret approach for comparing the security of programs.
A Hierarchical Security Assessment Model for Object-Oriented Programs

So far we have developed numerous security-related metrics, but having so many metrics to consider can be confusing for the programmer. In this chapter we present a hierarchical model of the metrics which can be easier to interpret. The model begins with our low-level security metrics based on characteristics of object-oriented classes, such as data encapsulation, cohesion and coupling. These metrics are then used to characterise higher-level properties concerning the overall readability and writability of classified data throughout a program. In turn, these metrics are then mapped to well-known security design principles such as ‘assigning the least privilege’ and ‘reducing the size of the attack surface’. Finally, the entire program’s security is summarised as a single security index value.

7.1 Introduction

Here we present a hierarchical model capable of assessing the overall security of a given object-oriented program based on a static analysis with regard to its security measured by our code-level metrics described in Chapter 6. The model consists of four hierarchical levels: (1) the potential flow of ‘classified’ data values between objects, (2) the overall readability and writability of classified data, (3) adherence to standard security design principles, and (4) a total assessment of the program’s security. Each of the top three levels aggregates metrics from the level below, allowing the programmer to view the program’s security at whatever level of
abstraction is desired. The metrics allow different versions of the same program, or different programs intended to do the same job, to be easily compared for their relative security.

Other models have been developed to quantify the overall security of programs. One of these is the ‘system vulnerability index’ which quantifies the overall security of a system based on its higher-level characteristics such as potentially neglectful acts [70]. Another is Alhazmi et al.’s approach which defines the security of a program based on its vulnerability density [127, 71]. To be effective, this technique depends on knowledge of the existing vulnerabilities a program might acquire. However, none of these approaches directly consider data-flow security of a program based on its overall architecture design, as we do.

Our approach in this chapter instead provides a hierarchical model (and supporting tool) for quantifying the security of whole object-oriented programs, taking into account the interactions between classes and the flow of classified data at the level of individual methods and even individual statements. (More details of our tool’s capabilities and architecture appear in Chapter 10.) In previous chapters we have defined metrics that measure how “classified” data values may flow between fields, methods and classes. We have also seen how specific changes to a program’s design may affect different metrics in different way. In this chapter we show how these low-level metrics can be aggregated to give the programmer an easy-to-understand assessment of the whole program’s security. (However, we admit that this convenience comes at the cost of precision, but that the detailed metrics can always be consulted if necessary.)

### 7.2 The Security Assessment Model

In this section, we explain how our hierarchical security assessment model provides a simple and transparent approach for assessing the relative security of object-oriented programs. The model can provide guidance for the development of secure programs using a bottom-up approach. It ensures that attention is given to lower-level detail in an object oriented program such as the number of security-critical attributes and methods. This in turn produces a measurement of a program’s security in terms of its higher-level data flow properties.

Our overall model for assessing the security of a given object-oriented program is shown in Figure 7.1. Each of its four levels defines a set of metrics for the security of a given program at a certain abstraction level. The bottom level defines numerous specific security metrics from the
7.2. The Security Assessment Model

The next level defines security from the perspective of the overall readability and writability of classified data. The next level up measures security with respect to the well-known security design principles reviewed in Chapter 2. Finally, the top level provides a single security measurement which summarises the total security of the entire program, allowing it to be compared easily with other similar programs. Each of the higher-level sets of metrics is defined based on those metrics at the level beneath it.

7.2.1 Metrics for the Potential Flow of Classified Data

The bottom level of the model shown in Figure 7.1 consists of our basic security metrics defined and described in Chapters 3, 4 and 6. Some measure obvious ways of directly reading from or writing to a classified attribute or field in a program. Others measure less obvious forms of data flow such as writing a classified value to an external device or file, serializing a class containing a classified value, or assigning a value to a classified attribute but not subsequently reading it (which allows a malicious programmer to add code to access the value without changing the program’s observable behaviour).
### Table 7.1: Readability and Writability Security Metrics

<table>
<thead>
<tr>
<th>Readability and Writability Metric</th>
<th>Name</th>
<th>Related Data Flow Security Metrics</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Readability of Classified Attributes</td>
<td>RCA</td>
<td>CIDA + CCDA + CAI</td>
<td>4</td>
</tr>
<tr>
<td>Writability of Classified Attributes</td>
<td>WCA</td>
<td>CMAI + CAAI + UACA</td>
<td>4</td>
</tr>
<tr>
<td>Readability via Classified Methods</td>
<td>RCM</td>
<td>COA + CME + CMI</td>
<td>3</td>
</tr>
<tr>
<td>Writability via Classified Methods</td>
<td>WCM</td>
<td>CAIW + CMW + CWMP + UCAM</td>
<td>3</td>
</tr>
<tr>
<td>Readability via Critical Classes</td>
<td>RCC</td>
<td>RPB + CPCC + CCE + CDP + CSP</td>
<td>2</td>
</tr>
<tr>
<td>Writability via Critical Classes</td>
<td>WCC</td>
<td>CCC + UCAC + CSCP + CSI</td>
<td>2</td>
</tr>
<tr>
<td>Security Absolute Measurements</td>
<td>SAM</td>
<td>CAT + CMT + CCT</td>
<td>1</td>
</tr>
</tbody>
</table>

#### 7.2.2 Metrics for the Readability and Writability of Classified Data

The large number of primitive metrics in Chapters 3, 4, and 5 can make it difficult for the programmer to gain an understanding of a program’s overall security. Therefore, the second level of metrics in our hierarchy summarises these metrics with regard to the essential properties of the readability and writability of classified data. The property of readability aims to define those lower level data flow metrics that allow reading of classified data to other parties either inside or outside the program. Similarly, the property of writability defines those lower level metrics that allow writing of classified data from other parties either inside or outside the program. As shown in Table 7.1, there are seven distinct metrics at this level: readability of classified attributes, writability of classified attributes, readability via classified methods, writability via classified methods, readability via critical classes and writability via critical classes, plus the absolute measurements.

These metrics sum the low-level, data-flow metrics within relevant, non-overlapping security classifications. This allows us to produce a smaller, more easily understood, set of security metrics which still contains sufficient information to give a clear measure of a program’s security. For example, data flow security metrics which are mainly concerned with measuring a classified attribute’s direct readability (i.e., CIDA, CCDA and CAI from Chapters 3 and 4) are mapped to the readability of classified attributes classification. Those security metrics which measure ways in which a classified data value can be accessed via a method that reads that attribute or reads from an external source of classified data (i.e., COA, CME and CMI from Chapters 3 and 4) are mapped to the readability via classified methods classification, and similarly for metrics relevant to the writability of classified data.
7.2. The Security Assessment Model

Table 7.2: Security Design Principle Metrics

<table>
<thead>
<tr>
<th>Security Design Principle</th>
<th>Name</th>
<th>Related Readability and Writability Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grant Least Privilege</td>
<td>PLP</td>
<td>WCA + WCM + WCC</td>
</tr>
<tr>
<td>Reduce Attack Surface</td>
<td>PRAS</td>
<td>RCA + RCM + RCC</td>
</tr>
<tr>
<td>Secure the Weakest Link</td>
<td>PSWL</td>
<td>WCA + WCM</td>
</tr>
<tr>
<td>Fail-Safe Defaults</td>
<td>PFSD</td>
<td>RCA + RCM</td>
</tr>
<tr>
<td>Least Common Mechanism</td>
<td>PLCM</td>
<td>RCC + WCC</td>
</tr>
<tr>
<td>Isolation</td>
<td>PI</td>
<td>WCC</td>
</tr>
<tr>
<td>Economy of Mechanism</td>
<td>PEM</td>
<td>SAM</td>
</tr>
</tbody>
</table>

via operations that access the critical class containing it (i.e., RPB, CPCC, CCE, CDP and CSP from Chapters 4 and 6). The other higher-level metrics aggregate those metrics whose main goal is to give a measurement of the direct writability of a classified attribute (i.e., CMAI, CAAI and UACA from Chapters 3 and 6), those that concern writability via classified methods (i.e., CAIW, CMW, CWMP and UCAM from Chapters 3 and 6), and those that measure writability via a critical class (i.e., CCC, UCAC, CSCP and CSI from Chapters 4 and 6). The seventh classification sums the absolute measurements which quantify the total number of classified attributes and methods, and the total number of critical classes in the entire program.

However, not all of these seven classifications may be considered equally important. Therefore, each sum is weighted as per the right-hand column in Table 7.1 so that metrics which measure direct accessibility of classified data are given higher weights than those that measure indirect accessibility of classified values.

7.2.3 Metrics for Specific Security Design Principles

This section presents the third level from the bottom in our pyramid-like hierarchical model. Numerous ‘security design principles’ have been published in the literature for developing and assessing security-critical systems [33, 11, 3, 32, 128]. Although these principles were generally not developed with software analysis in mind, we contend that several of them can be helpful in gaining an intuitive understanding of the overall security of a program. Therefore the metrics at this level group the readability and writability metrics to produce metrics relevant to particular design principles as shown in Table 7.2.

Seven relevant security design principles were chosen to measure the security of Java
programs: assign least privilege, reduce the size of the attack surface, isolation, economy of mechanism, use the least common mechanism, secure the weakest link and fail-safe defaults. (Further details of each principle can be found in Chapter 2.) Each of these was then quantified as the sum of relevant readability and writability metrics. In this case, however, some lower-level readability and writability metrics appear more than once since they cannot be uniquely mapped to the design principles.

We therefore reviewed each of these principles in the context of their significance for program design. This analysis showed that the least privilege principle is associated with all of the writability measurements while the attack surface principle is linked to all of the readability metrics. Furthermore, securing the weakest link is concerned with minimising the writability of security critical attributes and functions, and hence it is linked to the classified attributes and methods readability metrics. On the other hand, the fail-safe defaults principle aims to reduce the readability of attributes and functions, therefore it can be linked to the classified attributes and methods readability metrics. The least common mechanism principle is associated with the critical classes’ readability and writability metrics. However, isolation aims to reduce the amount of writability within classes, and hence it is associated with the critical classes writability metric. Finally, economy of mechanism’s main goal is to minimise the amount of critical data in the program. This means that the absolute security metric is the best one for this requirement. As shown in Table 7.2 the metric for each security design principle is then simply the sum of the relevant readability and writability metrics.

### 7.2.4 A Total Security Index

Finally, we define the highest level of our model to be a single Total Security Index (TSI) for a whole program. This index provides a simple way to compare the relevant security of entire object-oriented programs based on information obtained from lower-level security metrics. The TSI is simply the sum of the security design principle metrics, as shown in Table 7.3.

<table>
<thead>
<tr>
<th>Name</th>
<th>Metric</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSI</td>
<td>Total Security Index</td>
<td>PLP + PRAS + PSWL + PFSD + PLCM + PI + PEM</td>
</tr>
</tbody>
</table>

Of course, as we go higher in our hierarchy, the number of metrics is reduced, but they also
7.3 Experimental Results

This section demonstrates the validity of our security assessment model empirically. This was done using our software tool which is capable of calculating security metrics from compiled Java bytecode at all of the abstraction levels described above (Chapter 10). An example of its output is shown in Figure 7.2. It lists each of the metrics at all four levels and displays the design principles metrics graphically.

Figure 7.2: Example of Hierarchical Metrics Produced by the Java Bytecode Analyser for Apache James (1)

become less discerning. This is inevitable in such a model. In practice therefore, significant decisions should not be made on the basis of the more abstract metrics alone. Having appraised the overall situation using the higher-level metrics, the programmer should ‘drill down’ to the lower-level ones to understand the particular security issues affecting the program in question. (Our software tool described in Chapter 10 makes it easy to do this.) Further discussion of the challenges associated with combining disparate metrics can be found in Chapter 11.
Our hypothesis for testing the metrics is that related programs, developed for similar requirements and objectives, should produce better results as bugs are fixed and the program code is improved. In other words, the program’s Total Security Index (TSI) should decrease, indicating more secure code, from one long patch to the next (or at least should stay stable). However, we may also reasonably expect to see an increase in the TSI, indicating less secure code, if the program has major new functionality, and hence more potential vulnerabilities, added.

7.3.1 Approach

We conducted an experiment on five large-scale open source Java programs. We used our Java bytecode analyser (Chapter 10) to assess the relative security of different versions of the same program, since this makes the results meaningfully comparable. (Comparing the relative security of totally unrelated programs would not be informative, unless the actual security properties of the programs were already known to us.) The chosen projects were all security-related, so we could reasonably expect successive releases to be more secure than their predecessors. Following the process described in Chapter 6, we selected commonly-downloaded projects from SourceForge [126] but this time selecting different projects in order to further exercise the tool. A list of the open source programs was taken from Java Source [129] which provides some descriptions about different Java open source programs. For each project, we chose a specific version which was modified in a number of subsequent updates, to fix bugs found in the previous releases. In this way we could compare different versions of each program with identical functionality but (hopefully) improved code quality.

7.3.2 The Programs Analysed

The chosen programs consisted of the following: Apache James (two different programs) (releases 2.0.0, 2.1.0, 2.1.1, 2.1.2 and 2.1.3; and releases 2.2.0, 2.3.0, 2.3.1 and 2.3.2), SQL Jackcess (releases 1.0, 1.1, 1.2, 1.3, 1.4, 1.5 and 1.6), ACEGI (releases 1.0, 1.0RC1, 1.0RC2, 1.1, 1.2, 1.3, 1.4 and 1.5) and JGroups (releases 2.1, 2.2, 2.3 and 2.4).

Apache James was developed by the Apache Software Foundation to be a Java SMTP and POP3 Mail server and NNTP News server [130]. Jackcess is designed to work as a Java library for reading from and writing to MS Access databases [129]. ACEGI is a security program...
7.3. Experimental Results

designed to provide applications with comprehensive authentication, authorization, instance-based access control, channel security and human user detection capabilities [129]. JGroups provides several features mainly related to multicast communications using several networking transport protocols (e.g., TCP) [129].

7.3.3 Program Annotations

To let our bytecode analyser know which data values were considered security-critical we first needed to annotate the programs to identify their ‘classified’ data fields. We manually labelled a number of attributes in each project as ‘classified’, choosing attributes whose names and associated code comments indicated that they are most likely to store confidential data. We labelled the same attributes for each different release of the same program in order to make our comparisons meaningful.

For example, in the first release of Apache James, we annotated the following attributes as classified: userName, password and algorithm in the userDefault class. In addition to these attributes, we also labelled the following as classified in the later releases (2.2.0 onwards): fieldPassword and fieldUser in the Account class. With regard to the SQL Jackcess project, we annotated two different attributes as classified: SID and RESERVEDWORDS in the Database class. In the ACEGI project, we annotated three different attributes in two different classes to be classified: username and password in the ComparisonAttribute class, and password in the userAttribute class. In the JGroups project we chose the following attributes to be treated as classified: id and xid in the Xid class.

7.3.4 Analysis of Experimental Results

Given that lower values of each metric are considered more secure, programs whose metrics decrease are considered to have improved. We expected that these security-related programs would increase their security with each new release, except when significant new functionality has been added.

The results of applying our tool to each release of each project are summarised here for each of the four levels in Figure 7.1. The lowest level is the metrics for classified data flow. Given the
Table 7.4: Apache James 2.0.0 to 2.1.3 Data Flow Metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>2.0.0</th>
<th>2.1.0</th>
<th>2.1.1</th>
<th>2.1.2</th>
<th>2.1.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAT</td>
<td>7</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>CMT</td>
<td>52</td>
<td>57</td>
<td>57</td>
<td>57</td>
<td>37</td>
</tr>
<tr>
<td>CCT</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>CIDA</td>
<td>0</td>
<td>0.09090909</td>
<td>0.09090909</td>
<td>0.09090909</td>
<td>0.11111111</td>
</tr>
<tr>
<td>CCDA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>GOA</td>
<td>0.48076922</td>
<td>0.50877196</td>
<td>0.50877196</td>
<td>0.50877196</td>
<td>0.8108108</td>
</tr>
<tr>
<td>RPB</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CMAI</td>
<td>0.005383023</td>
<td>0.008528785</td>
<td>0.008528785</td>
<td>0.008528785</td>
<td>0.006146572</td>
</tr>
<tr>
<td>CAAI</td>
<td>0.013376137</td>
<td>0.009536785</td>
<td>0.009497965</td>
<td>0.009510869</td>
<td>0.004043127</td>
</tr>
<tr>
<td>CAIW</td>
<td>0.031850632</td>
<td>0.040585775</td>
<td>0.04033264</td>
<td>0.04029913</td>
<td>0.016741527</td>
</tr>
<tr>
<td>CMW</td>
<td>0.048237476</td>
<td>0.04022583</td>
<td>0.04022583</td>
<td>0.04022583</td>
<td>0.025928522</td>
</tr>
<tr>
<td>CWMP</td>
<td>0.115384616</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CPCC</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CCCC</td>
<td>0.01007326</td>
<td>0.007383479</td>
<td>0.007309274</td>
<td>0.007309274</td>
<td>0.007816862</td>
</tr>
<tr>
<td>CCE</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CME</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>UAC</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>UCAM</td>
<td>0.64</td>
<td>0.6753247</td>
<td>0.6753247</td>
<td>0.6753247</td>
<td>0.7407408</td>
</tr>
<tr>
<td>UCAC</td>
<td>0.25</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.6</td>
</tr>
<tr>
<td>CSP</td>
<td>0.33333334</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>CSI</td>
<td>0.002136752</td>
<td>0.005076142</td>
<td>0.005025126</td>
<td>0.005025126</td>
<td>0.002512563</td>
</tr>
<tr>
<td>CMU</td>
<td>0.2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.7</td>
</tr>
<tr>
<td>CAI</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CDP</td>
<td>0.025477707</td>
<td>0.02020202</td>
<td>0.02</td>
<td>0.02</td>
<td>0.025</td>
</tr>
<tr>
<td>CSCP</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.2</td>
</tr>
</tbody>
</table>

A large number of these metrics, we only present a representative example here. Table 7.4 shows the results for Apache James, versions 2.0.0 to 2.1.3. In these tables a green downwards arrow indicates the metric has decreased (i.e., security has improved) since the previous release, a red upwards arrow says that the metric has increased (i.e., security has worsened), and an amber right-pointing arrow means there has been no change to this particular metric. The results for Apache James (1) show that a large number of its lowest level metrics have increased in the second version when compared to the initial one. This is due to the increase of the total number of classified attributes and methods (as evidenced by the CAT and CMT counts). This indicates that new security-critical code was added to the program which has caused these metrics to increase, meaning that overall program security has gotten worse. However, the security metrics of the following two versions (i.e., 2.1.1 and 2.1.2) have managed either to decrease or stay stable which indicates an improvement in the overall security of the program. (No new security-critical code was added in these versions.) With regard to the last version, most of the metrics have decreased apart from those associated with measuring the proportion of classified methods.
which interact with classified attributes and number of critical classes (e.g. CCC, UCAM and CDP). Similarly detailed sets of data flow metrics were produced for the other four programs.

Table 7.5: Apache James (1) Readability and Writability Security Metrics

<table>
<thead>
<tr>
<th>Program</th>
<th>RCA</th>
<th>WCA</th>
<th>RCM</th>
<th>WCM</th>
<th>RCC</th>
<th>WCC</th>
<th>SAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>James_2.0</td>
<td>5.0423</td>
<td>2.5064</td>
<td>4.718</td>
<td>1.024</td>
<td>63</td>
<td></td>
<td></td>
</tr>
<tr>
<td>James_2.1</td>
<td>7.5263</td>
<td>2.267</td>
<td>6.04</td>
<td>1.524</td>
<td>72</td>
<td></td>
<td></td>
</tr>
<tr>
<td>James_2.2</td>
<td>7.5324</td>
<td>2.350</td>
<td>5.05</td>
<td>1.62</td>
<td>51</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.6: Apache James (2) Readability and Writability Security Metrics

<table>
<thead>
<tr>
<th>Program</th>
<th>RCA</th>
<th>WCA</th>
<th>RCM</th>
<th>WCM</th>
<th>RCC</th>
<th>WCC</th>
<th>SAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>James_2.2</td>
<td>7.376</td>
<td>1.443</td>
<td>72</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>James_2.3</td>
<td>7.051</td>
<td>1.008</td>
<td>108</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>James_2.3</td>
<td>7.051</td>
<td>1.008</td>
<td>108</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.7: SQL Jackcess Readability and Writability Security Metrics

<table>
<thead>
<tr>
<th>Program</th>
<th>RCA</th>
<th>WCA</th>
<th>RCM</th>
<th>WCM</th>
<th>RCC</th>
<th>WCC</th>
<th>SAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jackcess_1</td>
<td>6.190</td>
<td>0.110</td>
<td>45</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jackcess_1</td>
<td>6.190</td>
<td>0.093</td>
<td>72</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jackcess_1</td>
<td>6.190</td>
<td>0.108</td>
<td>53</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jackcess_1</td>
<td>6.190</td>
<td>0.105</td>
<td>90</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jackcess_1</td>
<td>6.190</td>
<td>0.105</td>
<td>90</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

At the next level of abstraction are the readability and writability security metrics as shown in Tables 7.5 to 7.9 for all five programs. Overall, most of the programs show a general improvement in their security metrics from one version to the next. This includes the results for
Chapter 7. A Hierarchical Security Assessment Model for Object-Oriented Programs

Table 7.8: ACEGI Readability and Writability Security Metrics

<table>
<thead>
<tr>
<th>Program</th>
<th>RCA</th>
<th>WCA</th>
<th>RCM</th>
<th>WCM</th>
<th>RCC</th>
<th>WCC</th>
<th>SAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACEGI_1.0</td>
<td>0</td>
<td>0.01753</td>
<td>5.78571</td>
<td>3.047665</td>
<td>6.02299</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>ACEGI_1.0_RC1</td>
<td>0</td>
<td>0.01682</td>
<td>6</td>
<td>3.02482</td>
<td>6.01194</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>ACEGI_1.0_RC2</td>
<td>0</td>
<td>0.01605</td>
<td>6</td>
<td>3.02352</td>
<td>6.0113</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>ACEGI_1.1</td>
<td>0</td>
<td>0.01735</td>
<td>6</td>
<td>3.04721</td>
<td>6.0229</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>ACEGI_1.2</td>
<td>0</td>
<td>0.01688</td>
<td>5.8</td>
<td>3.04754</td>
<td>6.0224</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>ACEGI_1.3</td>
<td>0</td>
<td>0.01505</td>
<td>5.8</td>
<td>3.04153</td>
<td>6.0194</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>ACEGI_1.4</td>
<td>0</td>
<td>0.01457</td>
<td>5.8</td>
<td>3.0405</td>
<td>6.0194</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>ACEGI_1.5</td>
<td>0</td>
<td>0.01523</td>
<td>5.8</td>
<td>3.042</td>
<td>6.0198</td>
<td>2</td>
<td>25</td>
</tr>
</tbody>
</table>

Table 7.9: Jgroups Readability and Writability Security Metrics

<table>
<thead>
<tr>
<th>Program</th>
<th>RCA</th>
<th>WCA</th>
<th>RCM</th>
<th>WCM</th>
<th>RCC</th>
<th>WCC</th>
<th>SAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jgroups_2.1</td>
<td>2.87290654</td>
<td>0.082377452</td>
<td>5.87274885</td>
<td>3.223844214</td>
<td>6.36905545</td>
<td>0.578395318</td>
<td>1247</td>
</tr>
<tr>
<td>Jgroups_2.2</td>
<td>2.86059322</td>
<td>0.08796402</td>
<td>5.862600228</td>
<td>2.890358991</td>
<td>6.36310332</td>
<td>0.645528986</td>
<td>659</td>
</tr>
<tr>
<td>Jgroups_2.3</td>
<td>3.414175054</td>
<td>0.058477185</td>
<td>6.37067205</td>
<td>2.915099895</td>
<td>6.73156628</td>
<td>0.662040983</td>
<td>869</td>
</tr>
<tr>
<td>Jgroups_2.4</td>
<td>3.097059418</td>
<td>0.046523018</td>
<td>6.32658255</td>
<td>3.057055518</td>
<td>6.69717832</td>
<td>0.782921134</td>
<td>1080</td>
</tr>
</tbody>
</table>

Apache James (1), versions 2.0.0 to 2.1.3. The only exception in this program is some of the readability and writability security metrics for the second release since this release introduced new security-critical features as mentioned previously, as clearly shown by the SAM metric. Another obvious exception is Apache James (2), versions 2.2.0 to 2.3.2, where only one metric (i.e. SAM) in version 2.3.0 has increased due to newly added security-critical code as well. However, the remainder of the metrics for both Apache programs decrease as the programs evolve, revealing an expected improvement in overall security.

One obviously exceptional program for the readability and writability security metrics is Jackcess (Table 7.7) which has shown an increase in the metrics associated with classified methods. This is clearly shown in the program’s WCM metric which has increased every time SAM increased. This is due to new classified methods since both WCM and SAM are related to measuring readability and writability via classified methods. However, once the program stabilizes, its security metrics decrease. This is clearly shown in the last two versions by a decrease in the security metric SAM indicating that there are no new security-critical features.
Table 7.10: Apache James (1) Security Design Principles Metrics

<table>
<thead>
<tr>
<th>Program</th>
<th>PLP</th>
<th>PRAS</th>
<th>PSWL</th>
<th>PFSD</th>
<th>PLCM</th>
<th>PI</th>
<th>PEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>James_2.0.0</td>
<td>3.605874835</td>
<td>9.75929375</td>
<td>2.58145481</td>
<td>5.04230766</td>
<td>5.742042118</td>
<td>1.024420024</td>
<td>63</td>
</tr>
<tr>
<td>James_2.1.0</td>
<td>3.866621133</td>
<td>13.93035628</td>
<td>2.34170189</td>
<td>7.8895224</td>
<td>7.565323283</td>
<td>1.524919243</td>
<td>72</td>
</tr>
<tr>
<td>James_2.1.1</td>
<td>3.86425309</td>
<td>13.92995224</td>
<td>2.33975651</td>
<td>7.8895224</td>
<td>7.564668799</td>
<td>1.524668799</td>
<td>72</td>
</tr>
<tr>
<td>James_2.1.2</td>
<td>3.864376395</td>
<td>13.92995224</td>
<td>2.339707596</td>
<td>7.88995224</td>
<td>7.564668799</td>
<td>1.524668799</td>
<td>72</td>
</tr>
<tr>
<td>James_2.1.3</td>
<td>4.01165019</td>
<td>13.02687684</td>
<td>2.390991341</td>
<td>7.97687684</td>
<td>6.67065885</td>
<td>1.62065885</td>
<td>51</td>
</tr>
</tbody>
</table>

Table 7.11: Apache James (2) Security Design Principles Metrics

<table>
<thead>
<tr>
<th>Program</th>
<th>PLP</th>
<th>PRAS</th>
<th>PSWL</th>
<th>PFSD</th>
<th>PLCM</th>
<th>PI</th>
<th>PEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>James_2.2.0</td>
<td>3.754880797</td>
<td>15.92986255</td>
<td>2.311881439</td>
<td>8.55384622</td>
<td>8.819015686</td>
<td>1.44299358</td>
<td>72</td>
</tr>
<tr>
<td>James_2.3.0</td>
<td>3.289438572</td>
<td>14.18750054</td>
<td>2.280825348</td>
<td>7.1359541</td>
<td>8.060159617</td>
<td>1.008613225</td>
<td>108</td>
</tr>
<tr>
<td>James_2.3.1</td>
<td>3.289381096</td>
<td>14.18750054</td>
<td>2.280767871</td>
<td>7.1359541</td>
<td>8.060159617</td>
<td>1.008613225</td>
<td>108</td>
</tr>
<tr>
<td>James_2.3.2</td>
<td>3.28900024</td>
<td>14.18750054</td>
<td>2.280387015</td>
<td>7.1359541</td>
<td>8.060159617</td>
<td>1.008613225</td>
<td>108</td>
</tr>
</tbody>
</table>

introduced, allowing an improvement in overall security. This also applies to ACEGI (Table 7.8) versions 1.2 to 1.5 with the exception of the last version (i.e., ACEGI 1.5) where two security metrics have increased. This version has worsened security with regard to its WCM and RCC metrics. Since the absolute SAM metric has not changed from its predecessor, this suggests that no security-critical code has been added to the program, so some non security-critical code must have been removed from the program increasing the relative proportion of critical code. This was confirmed by inspecting the program which showed that there was a decrease in the total number of non-critical classes. The program used to consist of 413 classes and in the last version it only contains 404.

By contrast, the JGroups programs (Table 7.9) show the most atypical result in this experiment as most of the security metrics continued to increase as the program evolved. The main reason for this is clearly shown by the SAM metrics which indicate that there is a significant increase in the number of security-critical features. For example, the data flow metrics of the JGroups 2.3 and 2.4 releases show that the number of classified attributes, i.e., those into which classified values may flow, has increased from 160 attributes in JGroups 2.3 to 208 classified attributes in JGroups 2.4.
### Table 7.12: SQL Jackcess Security Design Principles Metrics

<table>
<thead>
<tr>
<th>Program</th>
<th>PLP</th>
<th>PRAS</th>
<th>PSWL</th>
<th>PFSD</th>
<th>PLCM</th>
<th>PI</th>
<th>PEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jackcess_1.0</td>
<td>3.706</td>
<td>11.6</td>
<td>3.596</td>
<td>5.375</td>
<td>6.301</td>
<td>0.110</td>
<td>45</td>
</tr>
<tr>
<td>Jackcess_1.1</td>
<td>4.149</td>
<td>11.5</td>
<td>4.056</td>
<td>5.299</td>
<td>6.283</td>
<td>0.109</td>
<td>72</td>
</tr>
<tr>
<td>Jackcess_1.2</td>
<td>3.686</td>
<td>11.6</td>
<td>3.578</td>
<td>5.423</td>
<td>6.298</td>
<td>0.108</td>
<td>53</td>
</tr>
<tr>
<td>Jackcess_1.3</td>
<td>4.103</td>
<td>10.7</td>
<td>3.978</td>
<td>5.091</td>
<td>5.689</td>
<td>0.105</td>
<td>90</td>
</tr>
<tr>
<td>Jackcess_1.4</td>
<td>4.103</td>
<td>10.7</td>
<td>3.978</td>
<td>5.091</td>
<td>5.689</td>
<td>0.105</td>
<td>90</td>
</tr>
<tr>
<td>Jackcess_1.5</td>
<td>4.217</td>
<td>11.1</td>
<td>3.676</td>
<td>5.364</td>
<td>6.286</td>
<td>0.551</td>
<td>77</td>
</tr>
<tr>
<td>Jackcess_1.6</td>
<td>4.147</td>
<td>10.9</td>
<td>3.611</td>
<td>5.204</td>
<td>6.241</td>
<td>0.536</td>
<td>69</td>
</tr>
</tbody>
</table>

### Table 7.13: ACEGI Security Design Principles Metrics

<table>
<thead>
<tr>
<th>Program</th>
<th>PLP</th>
<th>PRAS</th>
<th>PSWL</th>
<th>PFSD</th>
<th>PLCM</th>
<th>PI</th>
<th>PEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACEGI_1.0</td>
<td>5.065</td>
<td>11.8</td>
<td>3.065</td>
<td>5.786</td>
<td>8.023</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>ACEGI_1.0_RC1</td>
<td>5.042</td>
<td>12.0</td>
<td>3.042</td>
<td>6</td>
<td>8.011</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>ACEGI_1.0_RC2</td>
<td>5.039</td>
<td>12.0</td>
<td>3.039</td>
<td>6</td>
<td>8.011</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>ACEGI_1.1</td>
<td>5.065</td>
<td>12.2</td>
<td>3.065</td>
<td>6</td>
<td>5.023</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>ACEGI_1.2</td>
<td>5.054</td>
<td>11.8</td>
<td>3.054</td>
<td>5.8</td>
<td>8.024</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>ACEGI_1.3</td>
<td>5.066</td>
<td>11.9</td>
<td>3.066</td>
<td>5.8</td>
<td>8.014</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>ACEGI_1.4</td>
<td>5.055</td>
<td>11.8</td>
<td>3.055</td>
<td>5.8</td>
<td>8.013</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>ACEGI_1.5</td>
<td>5.057</td>
<td>11.8</td>
<td>3.057</td>
<td>5.8</td>
<td>8.019</td>
<td>2</td>
<td>25</td>
</tr>
</tbody>
</table>

### Table 7.14: Jgroups Security Design Principles Metrics

<table>
<thead>
<tr>
<th>Program</th>
<th>PLP</th>
<th>PRAS</th>
<th>PSWL</th>
<th>PFSD</th>
<th>PLCM</th>
<th>PI</th>
<th>PEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jgroups_2.1</td>
<td>3.885</td>
<td>15.1</td>
<td>3.306</td>
<td>8.746</td>
<td>6.947</td>
<td>0.58</td>
<td>1247</td>
</tr>
<tr>
<td>Jgroups_2.2</td>
<td>3.623</td>
<td>15.1</td>
<td>2.978</td>
<td>8.723</td>
<td>7.008</td>
<td>0.645</td>
<td>659</td>
</tr>
<tr>
<td>Jgroups_2.3</td>
<td>3.636</td>
<td>15.1</td>
<td>2.973</td>
<td>9.784</td>
<td>7.395</td>
<td>0.662</td>
<td>869</td>
</tr>
<tr>
<td>Jgroups_2.4</td>
<td>3.886</td>
<td>16.1</td>
<td>3.103</td>
<td>9.426</td>
<td>7.480</td>
<td>0.782</td>
<td>1080</td>
</tr>
</tbody>
</table>
At the third level of abstraction we have metrics for the security design principles as shown in Tables 7.10 to 7.14. For instance, with respect to the security design principles of granting least privilege (PLP) and securing the weakest link (PSWL), the results in Tables 7.10 to 7.14 show that Apache James version 2.3.2 is the version which best satisfies these principles. For reducing the attack surface size (PRAS), Apache James version 2.0.0 is the version which best meets this requirement. Apache James version 2.0.0 also has the lowest metrics for readability of the program’s classified attributes and methods. This means that it is also the best with regard to the principle of fail-safe defaults (PFSD). For the design principle of the least common mechanism (PLCM), ACEGI version 1.1 best adheres to this principle. For the security design principle of isolation (PI), Jackcess version 1.1 is best. For economy of mechanism (PEM), which means minimising the amount of critical data, the ACEGI programs are the best among all the programs with regard to this absolute measure.

Finally, the top of our hierarchy is the Total Security Index. As shown in Table 7.15 for most cases the results confirm our assumption that in general the security metrics for the programs should improve as they evolve. For instance, both Apache James programs improve their TSI in all revisions of its code base except for their second one where a large amount of new code was added. The major exception is JGroups whose TSI shows a net increase indicating that its total security has worsened. This overall increase in TSI was primarily due to the security-critical functionality added in later releases. The other exception is ACEGI which has undergone a number of significant changes that affect security. Release ACEGI 1.0RC1 decreased the number of its classified methods while ACEGI 1.1 increased them by twelve. However, once the number of classified methods stabilised, the following releases showed the expected steady improvement in the TSI until the last one which had a slight increase in the number of interactions between methods and classified attributes.

### 7.4 Conclusion

This chapter has presented a hierarchical security assessment model which provides a simple and transparent approach for assessing the security of object-oriented programs at various levels of abstraction. The model takes into account low-level characteristics of an object-oriented program’s design, such as data encapsulation and cohesion, in order to measure higher-level characteristics such as the readability and writability of security-critical data. These are then
Table 7.15: All program’s Total Security Index (TSI)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>James 1</td>
<td>90.756229</td>
<td>109.11887</td>
<td>109.1134239</td>
<td>109.1133261</td>
<td>86.69771</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>James 2</td>
<td>112.812466</td>
<td>143.96249</td>
<td>143.962376</td>
<td>143.961615</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Jackcess</td>
<td>75.6539</td>
<td>103.36932</td>
<td>83.706322</td>
<td>119.657187</td>
<td>119.657188</td>
<td>108.18209</td>
<td>99.64834</td>
<td>-</td>
</tr>
<tr>
<td>ACEGI</td>
<td>59.747798</td>
<td>48.10716887</td>
<td>48.10173952</td>
<td>60.174849</td>
<td>60.773665</td>
<td>60.7519776</td>
<td>60.7488726</td>
<td>60.7541135</td>
</tr>
<tr>
<td>Jgroups</td>
<td>1285.577051</td>
<td>697.0659</td>
<td>909.96928</td>
<td>1120.79756</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

grouped to define another set of metrics which match the security design principles widely promoted in the literature. Finally, these values are used to quantify the overall security of the program of interest as a single metric.

This assessment can provide software developers with a simple way of comparing different versions of a program from the perspectives of classified data confidentiality and integrity. We have validated the model on several large-scale programs using our software tool which extracts the metrics from compiled Java bytecode automatically (Chapter 10). The case study analysed multiple versions of five programs to see how their security level changed. The results matched our intuitions about the way a program’s security changes as its code is either debugged, which should improve security, or extended, which may make security worse.

With this metrics hierarchy in place, we return in the next chapter to the problem of assessing the impact of refactoring on security (see Chapter 5), but this time at the level of program code.
Chapter 8

Security Assessment of Code-Level Refactoring Rules

In this chapter we assess the impact of a number of the most common code-level refactoring rules on security. To do so we ran our Java code analyser on various programs which were refactored according to each rule. New values of the metrics for the refactored programs then determine the impact of each rule on overall security.

8.1 Introduction

In Chapters 3 and 4, we have devised metrics for assessing the security of object-oriented designs and metrics that can measure the security of a given program at its code level as in Chapter 6. Here, we use these 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 using our metrics and then measuring the security of other refactored versions of the same program. The total security index (TSI) metric, which we have defined in Chapter 7, is used to provide a simple measurement of how a certain refactoring rule impacts security of the entire program based on information obtained from lower-level metrics. To make it easy to measure the security of various programs, we used our static analysis tool which can analyse compiled Java bytecode (Chapter 10).
8.2 Security Assessment of Refactoring Rules

Here we consider some standard refactoring rules which are applicable to an object-oriented program and identify their potential impact on the security of a given program when applied to security-critical parts of the code. (The similar study in Chapter 5 considered design-level refactoring only.) Table 8.1 lists the refactoring rules and their definitions [18, 87]. All of the chosen rules may have an impact on the size of a program’s ‘attack surface’ [40, 42, 43] and its adherence to the principle of granting ‘least privilege’ [33, 32, 2].

Applying the Add Parameter (AP) rule means that a classified attribute could potentially be passed as the method’s new parameter. If the method has another attribute whose value depends on the new parameter, this will then increase the number of classified attributes and subsequently the number of classified methods in the program. Conversely, removing a method’s parameter could potentially reduce the number of classified attributes and methods. Therefore, from this point of view, rule Add Parameter (AP) may decrease the security of a given program while Remove Parameter (RP) may increase security.

Refactoring rule Decompose Conditional (DC) aims to minimise the size of a conditional statement by extracting each case as a separate method [18]. In terms of security, if the decomposed case is a classified one (i.e., it reads or writes classified data), then the extracted method will be classified as well, and hence the refactoring will increase the number of classified methods. Therefore, applying this rule may decrease security, by making more parts of the program security-critical, and we should try to avoid decomposing security-critical conditional statements.

The refactoring rule of Replace Delegation with Inheritance (RDI) aims to reduce the coupling between two classes if a certain class (delegating) calls all or most of another class’s methods (delegate). This is done by replacing the delegation with inheritance by making the delegating class a subclass of the delegate class [87], which removes coupling between classes. With regard to this rule’s impact on security-critical features, it does not change the number of critical classes in a program. However, it will increase the number of security-critical superclasses in a program if the delegate class happens to be a critical one. This has both positive and negative effects on security. This allows subclasses to acquire more privileges over classified data which also increases the number of classified attributes and methods that could
## Table 8.1: Code-Specific Refactoring Rules

<table>
<thead>
<tr>
<th>Refactoring Rule</th>
<th>Identifier</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Parameter</td>
<td>AP</td>
<td>Add a parameter to an existing method to pass it extra information.</td>
</tr>
<tr>
<td>Remove Parameter</td>
<td>RP</td>
<td>Remove a parameter from a given method when it is not used.</td>
</tr>
<tr>
<td>Decompose Conditional</td>
<td>DC</td>
<td>Extract a method for each condition and then replace the condition with its corresponding method.</td>
</tr>
<tr>
<td>Replace Delegation with Inheritance</td>
<td>RDI</td>
<td>Create a subclass for the delegating class if it wants to inherit data from the delegate class.</td>
</tr>
<tr>
<td>Replace Inheritance with Delegation</td>
<td>RID</td>
<td>If the subclass does not inherit data from a superclass but its data is accessed by the superclass, then create a delegation between these two classes.</td>
</tr>
<tr>
<td>Replace Fields with Subclasses</td>
<td>RFS</td>
<td>Replace constant fields with subclasses for each field.</td>
</tr>
<tr>
<td>Replace Subclasses with Fields</td>
<td>RSF</td>
<td>Replace subclasses which have methods that return constant data with fields in a superclass.</td>
</tr>
<tr>
<td>Replace Array with Object</td>
<td>RAO</td>
<td>Replace an array which has elements of different types with an object with fields for each element in the array.</td>
</tr>
<tr>
<td>Replace Object with Array</td>
<td>ROA</td>
<td>Replace an object with an array with elements for every field in the object.</td>
</tr>
<tr>
<td>Replace Temp with Query</td>
<td>RTQ</td>
<td>Replace a temporary variable which holds a value of an expression with a method.</td>
</tr>
<tr>
<td>Replace Query with Temp</td>
<td>RQT</td>
<td>Replace a method which holds a value of an expression with a temporary variable.</td>
</tr>
<tr>
<td>Remove Setting Method</td>
<td>RSM</td>
<td>Remove a setting method for a field that only needs to set at the constructor level.</td>
</tr>
<tr>
<td>Parameterize Methods</td>
<td>PM</td>
<td>Replace methods that do similar operations depending on certain values with a method that takes a parameter representing these values.</td>
</tr>
<tr>
<td>Replace Parameter with Explicit Methods</td>
<td>RPEM</td>
<td>Replace a method which does similar operations depending on the value of its parameter with methods for each value of its parameter.</td>
</tr>
<tr>
<td>Replace Method with Object</td>
<td>RMO</td>
<td>Replace a long method whose local variables do separate things with a single class which has attributes for each of these local variables.</td>
</tr>
<tr>
<td>Replace Object with Method</td>
<td>ROM</td>
<td>Replace attributes of a class which are only called in certain methods with local variables inside these methods.</td>
</tr>
</tbody>
</table>
be inherited, making the program less secure. On the other hand, this rule improves security in another regard as an inheritance relation between security-critical classes reduces the links between them, and hence reduces the security coupling-based metric.

An exact contrast to the previous rule is the refactoring rule of Replacing Inheritance with Delegation (RID) which should be applied whenever a subclass does not inherit or inherits only parts of the superclass’s features [18]. In terms of the impact of this rule on security, again it has positive and negative effects. In the case where a class affected by this rule happens to hold some security-critical features then the rule will increase the number of links to a class with security-critical features, and it thus increases the coupling-based security metric. On the other hand, this rule improves security with regard to the security inheritance-based metrics by decreasing the number of security-critical superclasses in a program and the number of classified attributes and methods that could be inherited.

The refactoring rule of Replace Fields with Subclasses (RFS) helps to improve a program’s structure by introducing inheritance. It does this by replacing a class’s fields that are constants with subclasses in a way that does not affect the behaviour of the class [18]. With regard to the impact of this rule on a program’s security, it may create a critical subclass for each classified field, which increases the number of critical classes in a program, and hence makes it less secure.

The Replacing Subclasses with Fields (RSF) rule aims to reduce the number of classes in a program by removing those subclasses with methods which return constant values. This is done by replacing those methods with fields in the super class and deleting the subclasses [87]. In terms of this rule’s effect on security, it may reduce the number of critical classes, classified fields and classified methods, thus improving the overall security of a given program.

The refactoring rule of Replace Array with Object (RAO) aims to create a class for an array whose elements are of different types and make them fields in the new class [18]. In terms of the impact of this rule on a program’s security, this rule will make the program less secure if the replaced array contains classified data, because this means that the new class will have a classified field for every element in the array which means an increase in the number of critical classes, classified fields and classified methods. The new security-critical class will also be linked to the original class which will still be security-critical due to its continued reliance on classified data. Conversely, using the rule of Replace Object with Array (ROA) may increase
8.2. Security Assessment of Refactoring Rules

the security of a given program since it can reduce the number of critical classes, classified fields and classified methods.

The Replace Temp with Query (RTQ) refactoring rule’s main objective is to minimise the size of methods by replacing temporary variables that hold values of expressions with methods that return these values [87]. This rule in fact simplifies methods but it also increases their number. With regard to security, this can have the same effect as the Decompose Conditional rule as it might increase the number of classified methods if the values which are held by these local variables are classified. Thus, this rule may increase the number of classified methods and, as a result of this, worsen the security of a program. Conversely, the refactoring rule of Replace Query with Temp (RQT) aims to reduce the number of methods in a program by replacing methods’ return values with temporary variables in a certain method. The effect of this rule on security is the opposite to the Replace Temp with Query rule. This means that it may improve the security of a given program by reducing the number of its classified methods.

The Remove Setting Method (RSM) refactoring rule aims to reduce the number of methods in a given class. It allows us to remove a setting method for a field that only requires to be set at the creation time of that object [18]. In terms of this rule’s impact on security, if the removed method is a classified one then this reduces the number of classified methods in a class, and hence produces a more secure program.

The Parameterize Method (PM) refactoring rule is applied when there exist several methods which do similar things depending on certain values [87]. It allows us to combine these methods into a single one and then add a parameter to the new method that represents each of the previous methods, which eventually reduces the number of methods in a class [87]. If this rule is applied to methods that interact with classified attributes, then it will improve security by reducing the number of classified methods. This will also reduce most of the other security metrics, especially the cohesion-based and extensibility-based ones. On the other hand, the Replace Parameter with Explicit Methods (RPEM) refactoring rule aims to replace a method that does similar operations depending on the value of its parameter with a single method for each possible value of the parameter [87]. With regard to the impact of this rule on security, it may increase the number of classified methods if it is applied to a classified method. This increases the values of other security-based metrics such as cohesion and extensibility, and thus worsens the overall security of a program.
Refactoring rule Replace Method with Object (RMO) aims to decrease the length of a long method with many local variables which hold different values by replacing them with a single class to implement this method’s operations [87]. The new class will have a single attribute for each of the local variables while the old method has to call this class in order to complete its job [87]. In terms of the impact of this rule on the overall security of a program, this rule will increase the number of classified attributes, classified methods and critical classes if it is applied to a classified method which has local variables that hold classified values. This will increase the value of many of the security metrics and, therefore, worsen the overall security of such a program. On the other hand, the refactoring rule of Replace Object with Method (ROM) will decrease the number of classified attributes, classified methods and critical classes if it is applied to a class whose attributes are classified, and hence it makes the program more secure.

Thus we can see that each of the selected refactoring rules can potentially affect a program’s security, if applied to a security-critical code segment. Next we want to use our security metrics and analysis tool to confirm this empirically.

8.3 Case Study

The following case study illustrates how applying the refactoring rules shown in Table 8.1 impacts the security of a given program in a way measurable by our security metrics in Chapters 3, 4, 6 and 7, as predicted in Section 8.2. We expected the total security index (TSI) for each refactored program to match our prediction of the impact of these refactoring rules on security as described in Section 8.2. To support this, we used our tool for calculating the metrics from compiled Java bytecode (Chapter 10). An example of its output for one of the examined programs is shown in Figure 8.1. As usual, we began by labelling those attributes which may contain classified data as ‘classified’. The tool then identified all other attributes whose values may be influenced by ‘classified’ attributes automatically.

8.3.1 Original Program

The studied program for this work is inspired by refactoring examples presented by Fowler [131]. The program is a small bank loan system. It consists of four classes: Customer, Account, Loan and LoanType and is responsible for managing information
Figure 8.1: Example of Security Data Flow Metrics Produced by our Java Bytecode Analyser

about customers’ loans. Figure 8.2 shows a class diagram representing the relation between classes for this example.

The Customer class is responsible for storing information about a customer. This information consists of the customer’s name and the accounts used to manage their loans. It includes a classified attribute authentications for this class consisting of an array of strings that holds personal data identifying the customer. This authentication consists of a telephone number and password. In general, for privacy reasons, customers do not wish their personal and loans details to be made public.

The Account class holds information about the interest rate of a certain loan, which we also assume is classified. The Loan class is responsible for storing a loan’s details, including its ID and its type. The LoanType class holds attributes that describe the kind of loan, its name and its amount which represents a distinct value for each loan’s type. For this class, we assume that the loan amount is classified and thus it needs to be kept secret.
Figure 8.2: Bank Loan Program’s Original Design

8.3.2 Refactored Programs

This section illustrates the impacts of applying refactoring rules to the program described in Figure 8.2 on our security metrics. Most of these refactoring steps are inspired by similar ones from Fowler [131] and are defined in Table 8.1. These refactorings aim to make the program clearer and more maintainable, and are typical of the kinds of changes that a developer might reasonably contemplate. (The refactorings were applied to the Java program source code, but we have illustrated some using UML class diagrams for clarity.)

For instance, Figure 8.3 shows a refactored program using the Add Parameter rule applied to the original program. This refactoring step has changed the constructor of the Loan class from a default one that takes no parameter to a constructor that takes an integer as a parameter. Then it has created the loan attribute of type Loan in the Account class with a classified parameter.

Another refactored program is shown in Figure 8.4 which is refactored using the Decompose Conditional rule. This has been applied on the getStatement method in the Customer class. It has replaced the conditional cases with methods representing each case.
public class Account {

    @Marker("Classified") private int interestRate;

    /*
     * loan attribute before refactoring.
     */
    private Loan loan;

    public class Account {

        @Marker("Classified") private int interestRate;

        /*
         * loan attribute after refactoring using the
         * Add Parameter Refactoring rule.
         */
        private Loan loan = new Loan (interestRate);
    }

Figure 8.3: Bank Loan program refactored using Add Parameter

Figure 8.3 shows the design for a program refactored from the original one using the Replace Delegation with Inheritance rule. It has removed the attribute loanType of type LoanType and instead the Loan class has extended the LoanType class to inherit its features.

The rule Replace Fields with Subclasses has been used to produce the program design shown in Figure 8.6. It has been applied on the LoanType class to replace its constant fields with subclasses (i.e., Home, Car and Overdraft).

Another refactored design of the original program is shown in Figure 8.7. This version is refactored using the Replace Array with Object rule. The original Customer class has an array that is used to store the telephone number and the password of a customer, which are of different types. This refactored version has a new class to store these two attributes as shown in the Authentication class in Figure 8.7.

The refactoring rule of Replace Temp with Query has been applied to the original program to get a new version as shown in Figure 8.8. The affected method is getStatement in the Customer class. Its temporary variable thisAmount has been replaced with a method called getCharge in the Account class.

Figure 8.9 shows a revised version of the Customer class after applying the Remove Setting Method refactoring rule. This has resulted in removing all setter methods of
the authentications, and replacing them with a parameter in the Customer’s class constructor whose responsibility is to set the authentications attribute.

Figure 8.10 shows the result of applying the Parameterize Method rule on the setters and getters of authentications attribute in the Customer class. The rule has resulted in replacing the four methods that set and get the authentications attribute with two methods.
8.3. Case Study

### LoanType

- `final HOME` : int
- `final CAR` : int
- `final OVERDRAFT` : int
- «classified» `loanAmount` : int
- `name` : String

- `LoanType(LoanType _loanType)`
- `getLoanAmount() : int`
- `getName : String`

### Loan

- `loanID` : String

- `Loan(String _loanID)`

---

**Figure 8.5**: Bank Loan program design refactored using Replace Delegation with Inheritance

### LoanType

- «classified» `loanAmount` : int
- `name` : String

- `LoanType(String _name, int _loanAmount)`
- `getLoanAmount() : int`
- `getName : String`

---

**Figure 8.6**: Bank Loan program refactored design using Replace Fields with Subclasses
Chapter 8. Security Assessment of Code-Level Refactoring Rules

<table>
<thead>
<tr>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>- name: String</td>
</tr>
<tr>
<td>- accounts: Vector&lt;Account&gt;</td>
</tr>
<tr>
<td>- authenticate : Authentication</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Authentication</th>
</tr>
</thead>
<tbody>
<tr>
<td>- «classified» telephone : double</td>
</tr>
<tr>
<td>- «classified» password : String</td>
</tr>
</tbody>
</table>

| + Customer(String _name) |
| + getStatement() : String |
| + addAccount(Account arg) : void |
| + setAuthenticate(): void |
| + getAuthenticate : Authentication |

The result of applying the refactoring rule of Replace Method with Object on the getStatement method in the Customer class is shown in Figure 8.7. The result is a new class called Statement which has the same responsibilities of the original getStatement method in the Customer class. However, the original getStatement method now has to call the new method in order to implement its job.

### 8.3.3 Security Metrics Results

Table 8.2 shows the results of applying our security metrics to the original and refactored programs described in Section 8.3.2. The arrows indicate whether a metric has increased, stayed the same or decreased. Given that lower values of each metric are considered more secure, then we consider those refactoring rules which reduce these values, compared with those of the original program, to be refactoring rules that may make programs more secure.
8.3. Case Study

```java
// Temporary variable thisAmount before refactoring.
while (accounts.hasMoreElements()) {
    double thisAmount = 0;
    Account each = accounts.nextElement();
    switch (each.getLoan().getLoanType().getLoanAmount()) {
        case LoanType.HOME:
            thisAmount += 2;
            if (each.getInterestRate() > 2)
                thisAmount += (each.getInterestRate() - 2) * 1.5;
            break;
        case LoanType.CAR:
            thisAmount += each.getInterestRate() * 3;
            break;
        case LoanType.OVERDRAFT:
            thisAmount += 1.5;
            if (each.getInterestRate() > 3)
                thisAmount += (each.getInterestRate() - 3) * 1.5;
            break;
    }
    totalAmount += thisAmount;
    frequentCustomerPoints++;
    if ((each.getLoan().getLoanType().getLoanAmount()) == LoanType.OVERDRAFT) && each.getInterestRate() > 1) {
        frequentCustomerPoints++;
        result += "\t" + each.getLoan().getLoanType().getName() + "\t" + String.valueOf(thisAmount) + "\n";
    }
}
```

// Temporary variable thisAmount is replaced with
// getCharge() method using Replace Temp with
// Query refactoring rule.

```java
while (_accounts.hasMoreElements()) {
    Account each = _accounts.nextElement();
    totalAmount += each.getCharge();
    frequentCustomerPoints++;
    if ((each.getLoan().getLoanType().getLoanAmount()) == LoanType.OVERDRAFT) && each.getInterestRate() > 1) {
        frequentCustomerPoints++;
        result += "\t" + each.getLoan().getLoanType().getName() + "\t" + String.valueOf(each.getCharge()) + "\n";
    }
}
```

Figure 8.8: Bank Loan program refactored using Replace Temp with Query
Chapter 8. Security Assessment of Code-Level Refactoring Rules

---

**Figure 8.9:** Bank Loan program design refactored using Remove Setting Method

---

**Figure 8.10:** Bank Loan program refactored using Parameterize Method
(indicated by green, downward arrows “↓”). Refactoring rules which increase the value of the metrics are considered to be rules that may make programs less secure (red, upward arrows “↑”). Refactoring rules which do not change the value of the metrics indicate no effect on the program’s security level (indicated by amber, right arrows “→”).

For instance, applying the refactoring rule Replace Method with Object (RMO) has resulted in an increase for most of the security metrics. This includes all of the Absolute metrics, CAIW and CMW of the Cohesion-based metrics, UCAM and UCAC of the Extensibility-based metrics, and CDP of the Design size-based metrics. On the other hand, the values of CMAI and CAAI of the Cohesion-based metrics and Coupling-based metric CCC have decreased. In total, therefore, we say that applying the RMO rule has a negative impact on security because it increases the metrics overall.

However, if we apply the refactoring rule Replace Object with Method (ROM) to the
Table 8.2: Security Metrics Results

<table>
<thead>
<tr>
<th>Metric</th>
<th>Original</th>
<th>AP</th>
<th>DC</th>
<th>RDI</th>
<th>RFS</th>
<th>RAO</th>
<th>RTQ</th>
<th>RSM</th>
<th>PM</th>
<th>RMO</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAT</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>CMT</td>
<td>10</td>
<td>11</td>
<td>13</td>
<td>11</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>8</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>CCT</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>CIDIA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CCDA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>COA</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>RPB</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CMAI</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.175</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.233</td>
</tr>
<tr>
<td>CAAI</td>
<td>0.182</td>
<td>0.167</td>
<td>0.182</td>
<td>0.212</td>
<td>0.182</td>
<td>0.133</td>
<td>0.195</td>
<td>0.148</td>
<td>0.148</td>
<td>0.133</td>
</tr>
<tr>
<td>CAIW</td>
<td>0.429</td>
<td>0.5</td>
<td>0.429</td>
<td>0.474</td>
<td>0.429</td>
<td>0.5</td>
<td>0.435</td>
<td>0.368</td>
<td>0.368</td>
<td>0.52</td>
</tr>
<tr>
<td>CMW</td>
<td>0.667</td>
<td>0.688</td>
<td>0.722</td>
<td>0.786</td>
<td>0.619</td>
<td>0.667</td>
<td>0.688</td>
<td>0.615</td>
<td>0.615</td>
<td>0.706</td>
</tr>
<tr>
<td>CWMP</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CPCC</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CCC</td>
<td>0.222</td>
<td>0.25</td>
<td>0.222</td>
<td>0.222</td>
<td>0.167</td>
<td>0.2</td>
<td>0.222</td>
<td>0.222</td>
<td>0.222</td>
<td>0.2</td>
</tr>
<tr>
<td>CCE</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CME</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>UACA</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>UCAM</td>
<td>0.667</td>
<td>0.75</td>
<td>0.667</td>
<td>0.714</td>
<td>0.667</td>
<td>0.5</td>
<td>0.714</td>
<td>0.5</td>
<td>0.5</td>
<td>0.75</td>
</tr>
<tr>
<td>UCAC</td>
<td>0.333</td>
<td>0.333</td>
<td>0.333</td>
<td>0.333</td>
<td>0.333</td>
<td>0.25</td>
<td>0.333</td>
<td>0.333</td>
<td>0.333</td>
<td>0.5</td>
</tr>
<tr>
<td>CSP</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CSI</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.333</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CMI</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CAI</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CDP</td>
<td>0.75</td>
<td>0.75</td>
<td>0.75</td>
<td>0.75</td>
<td>0.429</td>
<td>0.8</td>
<td>0.75</td>
<td>0.75</td>
<td>0.75</td>
<td>0.8</td>
</tr>
<tr>
<td>CSCP</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TSI</td>
<td>56.36</td>
<td>59.06</td>
<td>59.69</td>
<td>70.87</td>
<td>70.45</td>
<td>59.37</td>
<td>57.91</td>
<td>52.42</td>
<td>52.42</td>
<td>63.19</td>
</tr>
</tbody>
</table>

Program in Figure 8.11 to the original program as in Figure 8.2, then we expect to see the opposite impact on the security metrics to the impact of Replace Method with Object (RMO). This means that ROM may decrease most of the metrics, and hence it may improve overall security of a program.

The final row of the table indicates how the refactoring rules affect the total security of a program based on their total security index values (TSI) from Chapter 7. From Table 8.2, we can see that the Add Parameter rule has increased most of the security metrics when compared with the original program, and hence applying the rule made the program less secure. On the other hand, the Remove Setting Method (RSM) rule’s total security impact shows a decrease in
most of these metrics, and hence this rule made the program more secure.

Of course the most secure program is one which has a lower value with regard to all of these security metrics. Unfortunately, the results show that we usually face a trade off because reducing one metric often results in increasing another (see Chapter 11).

### 8.3.4 Code Refactoring Rules Assessment

The last row of Table 8.2 shows the total impact of each refactoring rule on the overall security of the given program with regard to each security metric. An upward red “↑” means that the refactoring rule increased the total security index metric’s value for the examined program, and hence worsened its overall security. A downward green “↓” means that the refactoring rule decreased the values of the total security index metric, and hence improved security in this regard. An amber “→” indicates no effect of the refactoring rule on the program’s security.

The case study has confirmed our assumptions about how these refactoring rules would affect security as discussed in Section 8.3.2. For instance, the Add Parameter rule has worsened security since most of the metrics have increased when this rule is applied. This has been confirmed by its TSI when compared to the TSI of the original program. This also means that the Remove Parameter rule would improve security if it is applied to Figure 8.3 to get the original program which is shown in Figure 8.2.

This case is also applicable to other rules, including Decompose Conditional, Replace Delegation with Inheritance, Replace Fields with Subclasses, Replace Array with Object, Replace Temp with Query, Replace Parameter with Explicit Methods and Replace Method with Object. These rules have proved that they can increase many of the security metrics, so these rules, in addition to Add Parameter, will worsen the overall security of a given program if they are applied on security-critical code. Furthermore, the TSI of the programs which these refactoring rules are applied to confirm this prediction.

By contrast, the Remove Parameter, Replace Inheritance with Delegation, Replace Subclasses with Fields, Replace Object with Array, Remove Setting Method, Replace Query with Temp, Parameterize Methods and Replace Object with Method refactoring rules can improve security by decreasing many of the security metrics as predicted in Section 8.3.2. (These complementary rules are not shown in Table 8.2.) This is clearly shown by the case study
for Remove Setting Method and Parameterize Methods. However, other rules which have not been shown in this case study (i.e., Remove Parameter, Replace Inheritance with Delegation, Replace Subclasses with Fields, Replace Object with Array, Replace Query with Temp and Replace Object with Method) would have an opposite affect to worsen security as also shown by their relevant TSIs.

8.4 Conclusion

In this chapter we have studied the impact of sixteen refactoring rules on overall program security and illustrated this using a specific case study. This assessment has been carried out using our security metrics for measuring security with regard to potential information flow from Chapters 3, 4 and 6. Then the total security index (TSI) from Chapter 7 of each of the refactored programs in this study has confirmed our prediction of the effect of the rules on security for a given program. Out of the studied refactoring rules, it has been shown that eight could improve the overall security of a given program while the remaining eight may worsen its security. These results can serve to guide the application of refactoring rules to security-critical programs, alerting the programmer to potentially unsafe refactoring steps. Of course, a single case study is insufficient to produce entirely general conclusions about these rules, but given that the metrics confirm our assessment about the effect of the refactoring rules on security, we can have some confidence in the results.

As shown in Chapter 5, we can use certain refactoring rules as strategies for mitigating a number of API-level security vulnerabilities, and hence improve the overall security of a given program. This study has shown that there are a number of refactoring rules that can serve this purpose. For example, the Parameterise Methods refactoring rule helps mitigate the security vulnerability of accidentally leaking sensitive information by replacing methods that do similar operations depending on certain classified values with a single method that instead takes a parameter representing these values. This measurably improves the CAAI, CAIW and CMW security metrics. A similar example is to use the Remove Setting Method refactoring rule which removes a setting method for a field that only needs to be set at the constructor level. This rule can mitigate the security vulnerability of failing to protect stored data from being maliciously modified and hence makes the system more secure with regard to the CAAI, CAIW and CMW security metrics.
On the other hand, the study has shown that there are refactoring rules that need to be avoided in order to eliminate the possibility of introducing API-level security vulnerabilities. These include the Replace Fields with Subclasses rule which replaces constant fields with subclasses for each field. This rule can place security-critical data towards the top of the class hierarchy which allows unauthorised objects to maliciously inherit classified information through critical superclasses. This makes the program more vulnerable, and hence using such a refactoring rule could worsen security with regard to the CSP, CSI, CMI and CAI metrics.

As this chapter concludes the part of the thesis which studies the assessment of security-critical object-oriented programs, the next part illustrates the tool which we have developed to automatically extract our security metrics from the designs and code of object-oriented programs.
Part IV

AUTOMATED TOOL FOR SECURITY EVALUATION
A Type Inference System for Defining Secure Data Flow

To calculate our metrics we need to understand how information can flow between classified and unclassified variables (attributes, method parameters and return values) in object-oriented programs. In this chapter, we define a security type system consisting of a number of type inference rules which cover all of the relevant constructs for most of object-oriented languages. This type system formally defines the way in which our analysis tool traces information flow in order to measure the relative security of object-oriented programs.

9.1 Introduction

Type analysis is one of the most widely used approaches for verifying that the flow of security-critical information within a program is secure [132, 7]. In fact, type systems are capable of producing more reliable, readable and efficient programs in general [133, 134]. In particular, security type systems are used to check the correctness of security tools that are concerned with tracing information flow [135]. Examples of existing tools which were built to check information flow based on the source code include JFlow [135] and Jifclipse [136]. These tools provide a programming environment as an extension of a Java development environment, and aim to check the flow of ‘labelled’ classified information [135, 136]. They provide users with messages indicating the source of information leaks from labelled data to unlabelled ones by statically checking the program’s source code. They then compile the source code on a standard
Java compiler after removing the labels, which means that these labels are not included in the run-time code. Other projects have investigated information flow through computer program code, via type analysis [9] and data/control-flow analysis [10]. Notably Bernardeschi et al. [137] studied information flow in Java bytecode in order to develop more secure programs. Our code-level security metrics defined in Chapters 5 and 7 are capable of assessing the overall security of object-oriented programs. However, the tool which extracts these metrics (Chapter 10) needs to follow specific information flow principles to trace the security levels of program variables.

In this chapter we define a type inference system that characterises how our tool traces (potential) information flow when calculating security metrics for a Java program. Type inference has been widely used for defining valid information flow between variables in security-critical program code [9]. Our type system below follows the standard approach for imperative code [10], extended with rules for object-oriented program code [117].

### 9.2 A Security Type System

The main purpose of security type systems is to define the noninterference property, which ensures that classified data does not flow to unclassified variables [10, 117, 9]. A security type system is a collection of type inference rules, which describe how security levels are assigned to variables and statements in a given program [10], such that a typable program guarantees to secure explicit and implicit flows of classified information [138]. Explicit flows occur when the value of one variable is assigned to another, i.e., via the program’s data flow. Implicit flows occur when the value of one variable is used to control the execution of an assignment to another variable, i.e., via the program’s control.

This section illustrates the security type system which our tool implements in order to measure the potential flow of security-critical information. These rules define how our tool traces information flow through object-oriented code. The rules include judgments, which define the security type (level) of a program construct in a certain program context, of the following form.

\[ \text{context} \vdash \text{construct} : \text{type} \]

The security type can be either high-security or low-security (classified or unclassified), which we denote \( \text{hi} \) and \( \text{lo} \), respectively.
Each rule then has the form

\[
\frac{\text{hypotheses}}{\text{conclusion}} \quad \text{[name]}
\]

where the judgment appearing as the conclusion is considered valid if the hypotheses, if any, are all true. The hypothesis consist of other judgements on simple predicates. For instance, the \textit{if} rule in Figure 9.3 means that in order for conditional statement ‘\texttt{if } E \texttt{ then } S’ to be typed with security level $\tau$, it must be possible to type expression $E$ and statement $S$ as $\tau$, and the relevant part of the surrounding context must have security level $\tau$.

### 9.2.1 Type Inference Rules Contexts

This section explains the contexts in which program constructs are mapped to specific security levels in our rules. When typing a program construct the context is used to model significant properties of the program code surrounding the construct. Figure 9.1 shows the various sets used in our type inference rules’ contexts. We represent the context of a program construct as a quintuple $\langle \alpha, \beta, \gamma, \delta, \epsilon \rangle$.

- $\alpha : P \rightarrow L$, where $P = C \cup M \cup A$, is a function from a static labelling of classes, methods and attributes to security levels.

- $\beta : Q \rightarrow L$, where $Q = V \cup M$ which is a collection of the runtime variables and methods’ return types, is a function from instantiated objects, variables and methods to a ‘derived’ security level.

- $\gamma$ is the current security scope, either $\texttt{hi}$ or $\texttt{lo}$. This tells us whether or not the program construct is executed conditionally based on the value of a high-security variable. This is needed to detect implicit information flows [10, 117, 9].

- $\delta : M$ is the current method scope. It identifies the surrounding method, if any, and is used to determine the security level of the value returned by a method call.

- $\epsilon : Q \rightarrow C$ is a mapping from objects (variables) to their class. It is used to tell us from which class an object was instantiated, which in turn tells us the object’s (static) security level.
<table>
<thead>
<tr>
<th><strong>Levels</strong></th>
<th>$L = {\textit{classified, unclassified}}$</th>
<th>Set of security levels.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Statements</strong></td>
<td>$S = {S_1, \ldots, S_r}$</td>
<td>Set of program statements.</td>
</tr>
<tr>
<td><strong>Classes</strong></td>
<td>$C = {c_1, \ldots, c_n}$</td>
<td>Set of classes in the program.</td>
</tr>
<tr>
<td><strong>Method</strong></td>
<td>$M = {c_1.m_1, \ldots, c_p.m_p}$</td>
<td>Set of methods in the program, where for each method $c_p.m_p$, its class is $c_p \in C$.</td>
</tr>
<tr>
<td><strong>Attributes</strong></td>
<td>$A = {c_1.a_1, \ldots, c_i.a_i}$</td>
<td>Set of attributes in the program, where for each attribute $c_i.a_i$, its class is $c_i \in C$.</td>
</tr>
<tr>
<td><strong>Expressions</strong></td>
<td>$E = {E_1, \ldots, E_j}$</td>
<td>Set of expressions in the program, where expressions are constructed from variables, method calls, and binary and unary operators, i.e., $E ::= V</td>
</tr>
<tr>
<td><strong>Values</strong></td>
<td>$E_v = {E_{v1}, \ldots, E_{vh}}$</td>
<td>Set of expressions’ values in the program, excluding expression consisting of an object reference only.</td>
</tr>
<tr>
<td><strong>Variables</strong></td>
<td>$V = {v_1, \ldots, v_k}$</td>
<td>Set of runtime variables in the program which includes primitive expressions comprised of local variables, fields, objects and expression values.</td>
</tr>
<tr>
<td><strong>Fields</strong></td>
<td>$F = {f_1, \ldots, f_l}$</td>
<td>Set of fields in the program.</td>
</tr>
<tr>
<td><strong>Variable instance</strong></td>
<td>$X = {x_1, \ldots, x_y}$</td>
<td>Set of declared variable names that may appear in an expression.</td>
</tr>
<tr>
<td><strong>Object instance</strong></td>
<td>$O = {o_1, \ldots, o_z}$</td>
<td>Set of declared objects that may appear in executable code.</td>
</tr>
</tbody>
</table>

**Figure 9.1: Program Features Used in Contexts**

Together, functions $\alpha$ and $\beta$ define a security level of all relevant statically-declared and dynamically-instantiated constructs in the program, respectively. In effect, the purpose of the type inference rules is to show that these functions can be derived unambiguously from the program code and are consistent with the programmer’s annotations. Importantly, however, these functions may label more parts of the program as high-security than the programmer did, because our analysis may reveal information flows that were not noticed by the programmer.
9.2. A Security Type System

\[\alpha,\beta,\gamma,\delta,\epsilon \vdash E_1 : \tau \quad \alpha,\beta,\gamma,\delta,\epsilon \vdash E_2 : \tau\]
\[\alpha,\beta,\gamma,\delta,\epsilon \vdash (E_1 \oplus E_2) : \tau\]  \hspace{1cm} \text{[bi-operator]}

\[\alpha,\beta,\gamma,\delta,\epsilon \vdash E : \tau\]
\[\alpha,\beta,\gamma,\delta,\epsilon \vdash (\ominus E) : \tau\]  \hspace{1cm} \text{[uni-operator]}

\[\beta(E) = \tau\]
\[\alpha,\beta,\gamma,\delta,\epsilon \vdash (\text{new } C(E)) : \tau\]  \hspace{1cm} \text{[new-par]}
\[\alpha,\beta,\gamma,\delta,\epsilon \vdash (\text{new } C()) : \text{lo}\]  \hspace{1cm} \text{[new-def]}

\[\alpha,\beta,\gamma,\delta,\epsilon \vdash c : \tau\]
\[\alpha,\beta,\gamma,\delta,\epsilon \vdash ((C) o) : \beta(o)\]  \hspace{1cm} \text{[cast]}

\text{Figure 9.2: Type Inference Rules Defining How Data Flows Through Expressions}

9.2.2 Inference Typing Rules

This section explains the various rules which define a well typed system as shown in Figures 9.2, 9.3 and 9.4 (and are supported by the subtyping rules in Figures 9.5 and 9.6). If a certain program follows these rules, then it is well-typed, which in our case means that a consistent and allowable labelling of security levels can be applied to its objects, variables and fields statically. This means that all dynamic information flow inside this program will be secure as well. These rules consider Java language constructs relevant to flows of security-critical data. (They are also applicable to other object-oriented programming languages since they follow standard object-oriented terminologies and constructs.)

For instance, the bi-operator rule in Figure 9.2 types expressions that have a binary operator ‘\(\oplus\)’. The rule indicates that in order for a binary expression \(E_1 \oplus E_2\) to be typable at a specified security level \(\tau\), all of its enclosed sub expressions have to be typable at that level. Similarly, the uni-operator rule defines the type of expressions which have unary operators ‘\(\ominus\)’ preceding them to be the same as the expression \(E\).

The new-par rule is responsible for typing new expressions, instantiated from class \(C\), with a number of parameters \(E\). It ensures that this expression is only typable to be of security level \(\tau\) if the derived security level \(\beta(E)\) of the parameters to the class’s constructor is also \(\tau\). However, a new expression which has no parameters, as in rule new-def, means that this
\[
\begin{align*}
\gamma = \tau & \quad \alpha, \beta, \gamma, \delta, \epsilon \vdash E : \tau \quad \alpha, \beta, \beta(E), \delta, \epsilon \vdash S : \tau \\
\alpha, \beta, \gamma, \delta, \epsilon & \vdash (\text{if } E \text{ then } S) : \tau \\
\gamma = \tau & \quad \alpha, \beta, \gamma, \delta, \epsilon \vdash E : \tau \quad \alpha, \beta, \beta(E), \delta, \epsilon \vdash S_1 : \tau \quad \alpha, \beta, \beta(E), \delta, \epsilon \vdash S_2 : \tau \\
\alpha, \beta, \gamma, \delta, \epsilon & \vdash (\text{if } E \text{ then } S_1 \text{ else } S_2) : \tau \\
\gamma = \tau & \quad \alpha, \beta, \gamma, \delta, \epsilon \vdash E : \tau \quad \alpha, \beta, \beta(E), \delta, \epsilon \vdash S_1 : \tau \quad \alpha, \beta, \gamma, \delta, \epsilon \vdash S_2 : \tau \\
\alpha, \beta, \gamma, \delta, \epsilon & \vdash (S_1 ; S_2) : \tau \\
\gamma = \tau & \quad \alpha, \beta, \gamma, \delta, \epsilon \vdash E : \tau \quad \alpha, \beta, \beta(E), \delta, \epsilon \vdash S : \tau \\
\alpha, \beta, \gamma, \delta, \epsilon & \vdash (\text{switch } E \text{ case } S \text{ break}) : \tau \\
\gamma = \tau & \quad \alpha, \beta, \gamma, \delta, \epsilon \vdash S_1 : \tau \quad \alpha, \beta, \gamma, \delta, \epsilon \vdash S_2 : \tau \\
\alpha, \beta, \gamma, \delta, \epsilon & \vdash (\text{try } S_1 \text{ catch } S_2) : \tau \\
\alpha, \beta, \gamma, \delta, \epsilon & \vdash (\text{return } E) : \tau \\
\end{align*}
\]

Figure 9.3: Type Inference Rules Defining How Data Flows in Compound Statements

\[
\begin{align*}
\alpha, \beta[x \rightarrow \tau], \gamma, \delta, \epsilon[x \rightarrow T] & \vdash S : \tau \\
\alpha, \beta, \gamma, \delta, \epsilon & \vdash \{ T \ x ; \ S \} : \tau \\
\alpha(m) & = \tau \quad \alpha, \beta[p \rightarrow T_2, m \rightarrow T_1], \gamma, m, \epsilon \vdash S : \tau \\
\alpha, \beta, \gamma, \delta, \epsilon & \vdash \{ T_1 \ m \ (T_2 \ p) \ \{ S \} \} : \tau \\
\end{align*}
\]

Figure 9.4: Type Inference Rules Defining How Data Flows Through Declarations
expression is always typable with security level $l_0$. Note that the instantiated object itself is considered to be of low-security even if it contains high-security fields and methods. (However, if an object is passed as a parameter to a classified method, this object is considered to be of high-security, as per Figure 9.5).

The constant rule in Figure 9.2 says constants are always typable at any given security level. This is based on our assumption that constant literals do not inherently consist of classified or unclassified information. (Furthermore, Java bytecode cannot detect the flow of symbolic constants [117] so the tool in Chapter 10 cannot trace the flow of such constants). The cast rule says that cast expressions $(C) o$ always have the same security level $\beta(o)$ as the cast object $o$.

Figure 9.3 defines security levels for compound statements. In general, the security level for a compound statement is the maximum of that of its components [9]. This is done by requiring the compound statement and its components to have the same security level (see Figure 9.3), but allowing low-security constructs to be treated as high-security (Section 9.3) [9]. Conditional statements are represented in Figure 9.3 by the if, if-else, while and switch rules which ensure such statements are typable at a specific security level $\tau$ as long as their component conditions and branches are also all typable at that level [9].

Similarly, the compose rule states that in order for a block comprising two sequentially composed statements to be typable at a given security level $\tau$, both of the component statements have to be typable at that level. This is also true in the try rule where a ‘normal’ statement and its exception handler have to be typable at the same level as the whole try-catch block [118].

The compose rule types a block comprising two sequential statements at a specific security type. However, this does not take into account potential information flow arising from exceptions being thrown during sequential execution of statements. In the case when a condition causes an exception to be thrown during execution of the first statement it controls the execution of the second statement in the same way as a standard if statement condition. However, typing information flow via exceptions is a challenging task and in fact many security researchers admit this issue and do not include typing rules for exceptions [139] [140]. Similarly, we leave information flow due to exceptions for future work.

Lastly, the return rule states that a return statement in a method is only typable at level $\tau$ if the returned expression $E$ and the derived security level $\beta(\delta)$ for the surrounding method $\delta$ are also typable as $\tau$. 
Rules variable-declaration and method-declaration in Figure 9.4 are responsible for identifying newly-declared variables’ and methods’ types (i.e., parent classes). For a function \( f \), let \( f[d \rightarrow r] \) be that function with a new mapping from domain element \( d \) to range element \( r \), overriding any existing mapping if \( d \) is already in \( f \)’s domain. Rule variable-declaration, for typing a statement \( S \) in the scope of a declaration of variable \( x \) of type \( T \), requires statement \( S \) to be typed in a context where the derived security levels \( \beta \) are updated with a mapping from new variable \( x \) to security level \( \tau \) and variable \( x \) is known to be of program type \( T \) in function \( \epsilon \). Rule method-declaration, for typing the body \( S \) for a method \( m \) with return type \( T_1 \) and parameter \( p \) of type \( T_2 \), requires \( S \) to be typed in a context where parameter \( p \) is mapped to type \( T_2 \), method \( m \)’s value is mapped to return type \( T_1 \), the surrounding method context \( \delta \) is replaced by \( m \), and method \( m \) itself has security label \( \tau \).

### 9.2.3 Inference Subtyping Rules

So far, our rules require language constructs to have consistent security labels, either all \( \text{hi} \) or all \( \text{lo} \). For flexibility we therefore also need the ability to mix these labels where allowed. In particular, it is always safe to treat a low-security construct as high-security but not vice versa. To accommodate this we follow the usual approach of introducing subtyping rules that allow security labels for certain constructs to be ‘lifted’ from \( \text{lo} \) to \( \text{hi} \) [9, 119], as shown in Figures 9.5 and 9.6.

The subtyping logic associated with expressions is shown in Figure 9.5. For instance, the Low subtyping rule states that in order for variable ‘\( x \)’ to have a low-security (unclassified) level, the security level of its parent class \( \epsilon(x) \) and its derived security level \( \beta(x) \) have to both be unclassified. However, variable ‘\( x \)’ can be treated as high-security (classified) in any context as rule High states, since this always a safe assumption [10].

A similar approach is applicable to the rest of the expression subtyping rules including the subtyping rules for accessing fields (i.e., field(low) and field(high)) and invoking methods with or without parameters as shown in invoke-par, invoke-par-val and invoke-def. The only different rule is Invoke-par-ref which is concerned with invoking an object’s ‘\( o_1 \)’ method with another object ‘\( o_2 \)’ as a parameter which in Java means a reference to the object is passed. (Rule invoke-par-val is for parameter passing by value.) This rule states that in order for expression \( o_1.m(o_2) \) to be typable, at a high-security (classified) level, the derived security level of its passed object
9.2. A Security Type System

\[
\begin{align*}
\alpha(\epsilon(x)) &= \textbf{lo} & \beta(x) &= \textbf{lo} & [\text{Low}] \\
\alpha, \beta, \gamma, \delta, \epsilon &\vdash x : \textbf{lo} \\
\alpha, \beta, \gamma, \delta, \epsilon &\vdash x : \textbf{hi} \\

\beta(o) &= \textbf{lo} & \alpha(\epsilon(o), f) &= \textbf{lo} & [\text{field (low)}] \\
\alpha, \beta, \gamma, \delta, \epsilon &\vdash (o.f) : \textbf{lo} \\
\alpha, \beta, \gamma, \delta, \epsilon &\vdash (o.f) : \textbf{hi} \\

\beta(o) &= \textbf{lo} & \beta(m) &= \textbf{lo} & \beta(E) &= \textbf{lo} & [\text{invoke-par}] \\
\alpha, \beta, \gamma, \delta, \epsilon &\vdash (o.m(E)) : \textbf{lo} \\

\alpha, \beta, \gamma, \delta, \epsilon &\vdash (o.m(E_v)) : \textbf{hi} & [\text{invoke-par-val}] \\

\beta(o_2) &= \textbf{hi} & \beta(\epsilon(o_1).m) &= \textbf{hi} & [\text{invoke-par-ref}] \\
\alpha, \beta, \gamma, \delta, \epsilon &\vdash (o_1.m(o_2)) : \textbf{hi} \\

\alpha, \beta, \gamma, \delta, \epsilon &\vdash (o.m()) : \textbf{hi} & [\text{invoke-def}] \\
\end{align*}
\]

**Figure 9.5:** Inference Subtyping Rules for Expressions

\(\beta(o_2)\) and the derived security level of its invoked method \(\beta(\epsilon(o_2).m)\) have to be classified. This is done to allow for the possibility that the high-security method may ‘taint’ the pass-by-reference object with classified data.

Figure 9.6 shows the subtyping rules associated with assignment statements and statements generally. These rules allow low-security constructs to be treated as high-security, which is always safe, but not vice versa, which may be unsafe [10]. For instance, the \textit{f-assign} rules define the effect on security levels of assigning an expression \(E\) to a field or variable \(v\). The first part (low) says that an assignment to low-security variable \(v\) is typable only if the type of expression \(E\) is \textbf{lo} and if the assignment occurs in a low-security context. The latter constraint is necessary to prevent implicit data flow from a high-security variable that controls execution of this assignment to \(v\) [10]. The second part (high) says that we may assign an expression to a high-security variable and in any context. Rules \textit{ifs-assign} and \textit{sfs-assign} introduce similar constraints for assignments to instance fields of objects and static fields of classes, respectively.
\[ \gamma = \text{lo} \quad \beta(v) = \text{lo} \quad \alpha, \beta, \gamma, \delta, \epsilon \vdash E : \text{lo} \quad [\text{f-assign (low)}] \]
\[ \alpha, \beta, \gamma, \delta, \epsilon \vdash (v = E) : \tau \]
\[ \beta(v) = \text{hi} \quad [\text{f-assign (high)}] \]
\[ \alpha, \beta, \gamma, \delta, \epsilon \vdash (v = E) : \tau \]
\[ \gamma = \text{lo} \quad \alpha(\epsilon(o).f) = \text{lo} \quad \beta(o) = \text{lo} \quad \beta(f) = \text{lo} \quad \alpha, \beta, \gamma, \delta, \epsilon \vdash E : \text{lo} \quad [\text{ifs-assign (low)}] \]
\[ \alpha, \beta, \gamma, \delta, \epsilon \vdash (o.f = E) : \tau \]
\[ \beta(f) = \text{hi} \quad [\text{ifs-assign (high)}] \]
\[ \alpha, \beta, \gamma, \delta, \epsilon \vdash (o.f = E) : \tau \]
\[ \gamma = \text{lo} \quad \alpha(C.f) = \text{lo} \quad \alpha, \beta, \gamma, \delta, \epsilon \vdash E : \text{lo} \quad [\text{sfs-assign (low)}] \]
\[ \alpha, \beta, \gamma, \delta, \epsilon \vdash (C.f = E) : \tau \]
\[ \alpha(C.f) = \text{hi} \quad [\text{sfs-assign (high)}] \]
\[ \alpha, \beta, \gamma, \delta, \epsilon \vdash (C.f = E) : \tau \]

Figure 9.6: Inference Subtyping Rules for Statements

Finally, rule statement is a subsumption rule \[10\] which says that any statement \( S \) typable in a strict high-security context is also typable in a more lax low-security context.

### 9.3 Examples of Typing Security-Critical Code

In this section, we use our security type system to show its inference rules can be used to type security-critical programs. Our reason for defining the type system was to unambiguously document the necessary relationships between program constructs and security levels to be used when calculating our security metrics. More generally, such inference rules can be used to conduct proofs for valid variable typing. To illustrate this, hence we present proofs of some (very) small code fragments.
9.3. Examples of Typing Security-Critical Code

9.3.1 A Typable Code Fragment

Consider a program fragment compromising two assignments ‘\( m = l \)’ followed by ‘\( h = m \)’. Obviously such a fragment transfers the value of variable \( l \) to variable \( h \) via intermediate variable \( m \). Further assume that \( h \) is expected to be a high-security (classified) variable so in the context \( \beta(h) \) equals \( \text{hi} \). Figure 9.7 shows a proof using our rules that determines a set of conditions in which this code fragment is secure and typable as low-security. Recall that such a proof is constructed from the bottom up as a tree, with the judgements to be proved at the bottom and predicates needed to support the proof at the top. Each step up the tree is an instantiation of the one of the type inference rules. (Given the difficulty of displaying such trees, in Figures 9.7 to 9.10 we use an up-arrow ‘\( \uparrow \)’ to mark part of the proof which is displayed above the main tree.)

\[
\begin{align*}
\gamma &= \text{lo} \quad \beta(m) = \text{lo} \\
\alpha, \beta, \gamma, \delta, \epsilon &\vdash l : \text{lo} & \text{[Low]} \\
\alpha, \beta, \gamma, \delta, \epsilon &\vdash (m = l) : \text{lo} \\
\gamma &= \text{lo} \quad \uparrow \\
\beta(h) &= \text{hi} \\
\alpha, \beta, \gamma, \delta, \epsilon &\vdash (h = m) : \text{lo} & \text{[f-assign(high)]} \\
\alpha, \beta, \gamma, \delta, \epsilon &\vdash (m = l \quad ; \quad h = m) : \text{lo} & \text{[compose]} 
\end{align*}
\]

**Figure 9.7**: A Proof for Typable Sequence of Assignments

Reading from the bottom up, the first step in Figure 9.7 uses the sequential composition rule to determine that the program fragments context \( \gamma \) must be low-security if the fragment is to be typable as low. The *f-assign (low)* and *Low* rules are then applied to assignment ‘\( m = l \)’ to determine that both variables \( l \) and \( m \) can be typed as low-security in function \( \beta \) provided that the class (type) from which variable \( l \) was instantiated is not critical (i.e. was not declared as high-security). On the right of the proof tree the *f-assign (high)* rule is used on assignment ‘\( h = m \)’ to determine that this assignment is secure if we derive \( h \) to be a high-security variable all of which is consistent with our original assumptions about the programmer’s annotations.
9.3.2 A Non-Typable Code Fragment

This example in Figure 9.8 shows an attempted proof of a similar program to the previous one. However, in this case the order of the assignments has been reversed. Furthermore, we assume that \( h \) is a field \( f \) of an object \( o \) which has been annotated in its present class \( C \) by the programmer as classified. Thus, the context includes the fact that \( \alpha(o.f) \) equals \( \text{hi} \). Figure 9.8 shows one attempt to prove that this code is type-correct in a low-security context.

\[
\begin{align*}
\gamma &= \text{lo} \quad \beta(m) = \text{lo} \\
\beta(o) &= \text{lo} \\
\alpha(o.f) &= \text{lo} & \text{[field (low)]} \\
\alpha, \beta, \gamma, \delta, \epsilon \vdash h : \text{lo} & \text{[f-assign (low)]} \\
\alpha, \beta, \gamma, \delta, \epsilon \vdash (m = h) : \text{lo} \\
\gamma &= \text{lo} \\
\gamma &= \text{lo} \quad \beta(m) = \text{lo} \\
\alpha(e(l)) &= \text{lo} \\
\beta(l) &= \text{lo} & \text{[Low]} \\
\alpha, \beta, \gamma, \delta, \epsilon \vdash l : \text{lo} & \text{[f-assign (low)]} \\
\alpha, \beta, \gamma, \delta, \epsilon \vdash (l = m) : \text{lo} & \text{[compose]} \\
\alpha, \beta, \gamma, \delta, \epsilon \vdash (m = h) & \quad l = m : \text{lo}
\end{align*}
\]

**Figure 9.8:** Attempted Proof for Non-Typable Sequence of Assignments

Since this program consists of a block comprising two sequentially composed statements, it has to be typed according to the \textit{compose} rule. Therefore, in order for this program to be typable in a low-security context, each one of its assignment statements has to be typable at low-security context as well. The second assignment \( l = m \) is typed using the \textit{f-assign} and \textit{Low} rules in such a way that variable \( l \) and \( m \) are of low-security. However, the attempt to type the first assignment \( m = h \) as a low-security assignment using the \textit{f-assign (low)} and field rules requires us to show that \( \alpha(o.f) \) equals \( \text{lo} \) which contradicts the programmer’s annotations. Alternatively, we could have applied the \textit{f-assign (high)} rule to the first assignment, but this would require us to show that \( \beta(m) \) equals \( \text{hi} \), which contradicts the requirement for the second assignment that variable \( m \) only contains low-security data. In fact, there is no way to correctly type this code fragment in the given context. Therefore, Figure 9.8 shows that this program is not typable according to our type inference rules in a low-security context.
9.3.3 Typing A Control Flow Expression

The previous examples considered explicit data flow. To illustrate the effect of implicit, high-security control flow, hence we type a conditional statement. Consider the conditional statement at the bottom of Figure 9.9. Based on the value of variable $h$ it updates the value of either variable $h$ or $l$. Under the assumption that variable $h$ is known to be of high security, i.e., $\beta(h)$ equals $hi$, hence we conduct a proof that reveals any other conditions necessary for this to be valid code.

![Figure 9.9: Typing A Control Flow Expression](image)

For a conditional statement to be typable at a specific security level, all of its conditions and branches have to typable at that level, as the if-else rule states. Therefore, for the conditional statement shown in Figure 9.9 to be typable at security level $hi$, its conditions and branches have to be typable at that level. Figure 9.9 shows that this statement’s condition is typable as high security according to the bi-operator rule. Its first branch, which is a variable assignment, is also typable at a high-security level, and introduces the necessary assumption that $h$ is of high security. Typing the second branch follows similarly but this time introduces the requirement that variable $l$ must also be high security, regardless of the security level of source variable $y$. Even if the programmer has intended variable $l$ to be a low-security variable (or field), the type inference rules require that it is ultimately labelled as high security because its value is updated in a context under the control of high-security expression ‘$h > 0$’.
Figure 9.10: An Attempt to Type an Object-Oriented Program Fragment

9.3.4 Typing An Object-Oriented Program Fragment

The previous examples have illustrated data-flow and control-flow analysis in our type system. Figure 9.10 shows an example of an object-oriented program fragment, and how it is typed using our type inference rules. It consists of a block comprising four sequentially composed statements. The first statement shows an object \( o_1 \) instantiated from class \( C \). The second statement is an assignment whose target variable \( l_1 \) is assigned the value of a field from object \( o_1 \). The third statement invokes a method \( m \) of object \( o_2 \) using object \( o_1 \) as a parameter. The fourth statement is another assignment in which the value of variable \( l_2 \) is assigned from a field of object \( o_1 \). For this example, we assume that the declared security level of field \( l \) is low, i.e., \( \alpha(C.l) = \text{lo} \) and the derived security level of method \( m \) is high, i.e., \( \beta(m) = \text{hi} \).

As per the compose rule, all four of these statements have to be typable at the same level, here assumed to be low. It can be seen that the first two statements are typable at this level as stated by the new-def and f-assign rules. Similarly, the fourth statement is typable by treating
object $o_1$ as low security. However, the proof of the third statement requires the security level of object $o_1$ to be high because it is used in as a pass-by-reference parameter to high-security method $m$, as per the \textit{invoke-par-ref} rule, and hence it is only typable at a high-security level. There is thus a contradiction between the necessary derived security level for object $o_1$ which is required to be low security by statements two and four but high-security by the third statement. This reveals the ‘tainting’ of this object. (To correctly type this program fragment we therefore obliged to ‘lift’ the security level of object $o_1$ to \texttt{hi}.)

\section*{9.4 Conclusion}

Due to the importance of type systems in defining information flow through security-critical programs, this chapter presented a secure flow type system to define the principles that our tool uses in order to trace security levels of program variables when calculating our security metrics. Our tool, which is described in the next chapter, implements on these principles to identify the various ways of potential flow of classified data within a given program.
An Automated Tool for Assessing Security-Critical Designs and Programs

This chapter describes in detail our Security-Critical Program Analyser (SCPA). SCPA is used to assess the security of a given program based on its design and source code with regard to our security design and code level metrics. Furthermore, it allows software developers to generate a UML-like class diagram of their programs and annotate its confidential classes, methods and attributes. SCPA is also capable of producing Java source code for the generated design of a given program. This source code can then be compiled and the resulting Java bytecode program can be used by the tool to assess the program’s overall security based on our security code metrics.

10.1 Introduction

Most existing security analysis tools aim to assess a program’s security with regard to the existence of previously defined vulnerabilities. These tools search for source code bugs to analyse the security of a given program. Examples of such tools include FindBugs [141][142], PMD [143] and Checkstyle [144] which are developed to identify any potential vulnerabilities in Java programs. In fact, most tools developed for this purpose concentrate on finding vulnerabilities in non object-oriented programs such as programs developed in C. An example of these tools is MOPS [145] developed by Chen and Wagner, which is a model checking security program to identify source code bugs such as abuses of setuid bugs. However, both
of these types of tools cannot evaluate overall security of security-critical programs. A more useful tool would be one which measures the overall security of a given program.

Those tools which quantify a program’s quality are, in most cases, not security oriented. In other words, they do not consider security to be part of the overall program’s quality and therefore they do not measure the security level of a given program. Some examples of these tools are: OOMeter \cite{146} which measures program quality based on UML designs; JHawks \cite{147}, JMetric \cite{148}, Metrics \cite{149} and *J \cite{150} which extract a number of static quality metrics from source code and DynaMetrics \cite{151} which uses existing object-oriented quality metrics to dynamically analyse and evaluate quality of Java programs.

Other tools such as Fujaba \cite{152,153}, UMLet \cite{154}, UMLGraph \cite{155,156} and ArgoUML \cite{157} enable developers to design programs using UML diagrams. Fujaba is further capable of generating Java source code of a given UML design \cite{152,153}. However, none of these tool are security-oriented and hence do not annotate confidential data or assess the security of a given design. Therefore, a tool that is capable of quantifying the overall security of a program at various stages during its development would be ideal.

This chapter describes our Security-Critical Program Analyser (SCPA) which allows software developers to design and assess the overall security of their security-critical programs easily. SCPA allows developers to create designs which annotate confidential data (i.e., classes, methods and attributes) and define interactions between methods and attributes. Such designs can then be analysed to extract our object-oriented security design metrics. From this design, Java source code and Java bytecode can be produced. The resulting Java bytecode can be analysed to extract our security metrics for object-oriented programs to give more reliable results about the program’s overall security.

Software developers can also use SCPA to edit designs of existing security-critical programs using our security refactoring rules defined in Chapters 5 and 8 in order to improve their security. In this chapter, we show in more detail the capabilities of our tool using a case study consisting of a small security-critical program.
10.2 Security-Critical Program Analyser (SCPA) Architecture

This section describes our tool’s architecture for designing security-critical programs, generating Java source code for an existing security-critical design, and assessing the security with regard to our design and code level metrics. SCPA consists of sixteen different classes, more than thirteen thousand of lines of code and four different tools. These are meant to achieve the main goal of SCPA, which is to automatically assess the overall security of security-critical designs and programs in an easy way. SCPA’s sub-tools include: a UML Design Security Analyser (UMLDSA), a Java Source Code Generator (JSCG), an External Java Compiler and a Java Bytecode Security Analyser (JBSA). Figure 10.1 shows the sub-tools which are part of SCPA and their outputs, which are described in more detail below.

Figure 10.1: Security-Critical Program Analyser (SCPA) Architecture

10.2.1 UML Design Security Analyser (UMLDSA)

This part of the tool is responsible for providing programmers with two important features for designing security-critical programs as shown in Figure 10.2. One feature is the ability to generate annotated UML-like class diagrams with UMLsec and SPARK’s annotations. The second feature is the ability to evaluate the security of these diagrams with regard to our security design metrics. These features are illustrated by Figure 10.3, which shows the UMLDSA main interface and design display area.
Chapter 10. An Automated Tool for Assessing Security-Critical Designs and Programs

**UML DESIGN SECURITY ANALYSER (UMLDSA)**

1. **Generate UML Security-Critical Class Diagram**
   - Store the class signature and relations (e.g., criticality and extensibility)
   - Store the class attributes and their characteristics (e.g., classified and accessibility)
   - Store the class methods and their characteristics (e.g., interactions, accessibility, and extensibility)
   - Draw the class and store it to the linkedhashmap containing the design’s classes details

2. **Assess UML Security-Critical Class Diagram**
   - Analyse each class characteristics (e.g., criticality and extensibility)
   - Analyse each attribute characteristics (e.g., classified and accessibility)
   - Analyse each method characteristics (e.g., interactions, accessibility and extensibility)
   - Analyse relation between classes (e.g., coupling and inheritance)
   - Calculate & Display security design metrics (e.g., coupling, cohesion and extensibility)

**Figure 10.2**: UML Design Security Analyser (UMLDSA)

**Figure 10.3**: UMLDSA User Interface
10.2. Security-Critical Program Analyser (SCPA) Architecture

(a) Classes Interface

(b) Attributes Interface

(c) Methods Interface

(d) Libraries Interface

Figure 10.4: New Class Details User Interfaces
Generate UML Security-Critical Class Diagram: As shown in Figure 10.2, this part of the tool is responsible for producing the UML-like security-critical class diagram. Figure 10.4 shows the tool’s user interface for generating a single class in the design. This process involves four major steps:

1. Storing the class signature and relations. The class signature defines whether the class is critical (i.e. contains classified attributes), extensible (i.e. not final), a superclass or subclass. The signature also defines the class’ accessibility (e.g., public or private). This step also involves storing the relations of this class with others in the design. This includes inheritance, association and composition relations. (We are aware that there are other types of relations between classes but these are the only ones required by our security metrics.)

2. Storing the class attributes and their details. This includes storing the name of the attribute, type and accessibility in addition to whether the attribute is classified, static and/or final.

3. Storing the class methods and their details. The details required for each method are: the method signature which includes name, type, accessibility, parameters, extensibility, and if it is static; whether the method is classified, and the attributes which the method interacts with and how it interacts with them (i.e., setting and/or reading their values).

4. Storing the Java libraries required by the design. This step is not necessary for assessing the security of a UML class diagram, but is required when converting the design to Java source code.

When all of the information for a class has been entered, the class information is saved and the class is added to the diagram in the UMLDSA display area.

An existing class in the design can be edited. Attributes and methods can be added, deleted or changed using the same interface for class entry (see Figure 10.4). A class can also be deleted from the design using the interface in Figure 10.5.

A UML design may be saved to a text file. The tool can also import designs from text files. The interface for these operations are shown in Figure 10.6.
Figure 10.5: Deleting a UML Class Interface

Figure 10.6: Saving and Opening UML Design Interfaces
Assess UML Security-Critical Class Diagram: Once a design has been created or loaded, it can be assessed according to our security design metrics, as described in Chapters 3 and 4. The analysis is carried out as follows:

1. Each class in the design has its attributes and methods inserted into a number of different sets. Any attribute or method may belong to more than one set.

2. The attributes of a class are added to sets for: classified attributes (i.e. labelled “secrecy”), non private static classified attributes, non private instance classified attributes, classified attributes which can be inherited (i.e., classified attributes in a critical superclass) and classified attributes in inheritance hierarchy (i.e., classified attributes in a critical superclass or critical subclass).

3. The methods of a class are divided into sets of: classified methods (labelled “secrecy”), non private classified methods, extensible classified methods, classified methods which can be inherited (i.e., classified methods in a critical superclass) and classified methods in an inheritance hierarchy (i.e., classified methods in a critical superclass or critical subclass), classified attributes that are set by mutators and/or read by accessors.

4. The class itself is then added to other sets depending if it is: critical (i.e., contain classified attributes), extensible, superclass and subclass.

5. Relationships between classes are also identified and classes are added to sets depending on which classes are coupled to other classes and though which attributes, which critical classes are composed-part private classes, and which classes are in an inheritance hierarchy.

6. All of these set are then used to calculate our security design metrics as follows.

   - Sets which contain information related to the size of classified attributes, non private static classified attributes, non private instance classified attributes, classified methods and non private classified methods are used to calculate the data encapsulation-based security metrics.
   - Sets which contain information related to the size of classified attributes, mutated classified attributes, accessed classified attributes, mutators, accessors, all methods and all classified methods are used to calculate the cohesion-based security metrics.
• Sets of all classes, critical classes, mutated classified attributes and accessed classified attributes in the design are used to calculate the coupling-based security metric.

• Sets of all critical classes and composed-part private critical classes in the design are used to calculate the composition-based security metric.

• Sets which contain information related to the size of critical classes, classified methods, extensible critical classed and extensible classified methods are used to calculate the extensibility-based security metrics.

• Sets of all classes, critical classes, critical superclasses, classified methods in inheritance hierarchies, inheritable classified methods, classified attributes in inheritance hierarchies, inheritable classified attributes are used to calculate the inheritance-based security metrics.

• Sets of all classes and critical classes in the design are used to calculate the design size-based security metric.

10.2.2 Generating Java Source Code (JSCG)

Another main feature of our tool is the ability to generate the relevant Java source code from UML Security-Critical Class Diagrams as a basis for calculating design-level metrics. This generated code includes those attributes which the user labelled to be “Classified”. To generate the body of each method in a given class, the tool analyses the SPARK annotations generated by the UMLDSA that identifies which attributes are mutated or accessed by the class’ methods.

As shown in Figure 10.7, this part of the tool is responsible for creating a Java program based on the output of the UML Design Security Analyser (UMLDSA). UMLDSA produces a map that contains all the design’s classes and their attributes, methods, and imported libraries. JSCG produces the program’s equivalent Java source code, which can be compiled and analysed to assess the security of the program at the code level.

From the design, a number of Java source code files are created and saved to a specific location (Figure 10.8). First of all a file called Marker.java is generated. This Java class is responsible for producing the annotations used to label security-critical attributes. Next a new Java file is created for each class in the design. If a class is nested (inner class) then it is included
in the Java file with its enclosing class. A class is written to the file as follows:

1. Import statements are written for any libraries that were identified as being required by the class.

2. The class signature is then written, including the class name, accessibility and any superclasses.

3. Next each attribute for the class is declared and annotated, including the the attribute name, accessibility, type and if this is a class or instance attribute.

4. Lastly the methods for each class are declared. The method signature is written follows
by the method body which shows which attributes that are mutated or accessed by this method.

10.2.3 Java Source Code Compiler

The third feature of our SCPA is to provide developers with the ability to compile their Java source code programs. Figure 10.9 shows the interface for this feature. This is used to either compile Java code skeletons generated from UML specifications, for calculating design-level metrics, or for compiling annotated Java programs, for calculating code-level metrics.

This part of our tool uses an external compiler developed as a part of the Eclipse Java Development Tools (JDT) [158]. The compiler has been configure to compile annotated Java programs and integrate these annotations into the compiled Java bytecode. The compiler has also been set to display information about the compiled files such as their number, number of lines of code and the time taken to compile. The tool will display any compilation errors given by the compiler in the text area shown in Figure 10.9.
10.2.4 The Java Bytecode Security Analyser (JBSA)

The final feature of our tool is the JBSA which can automatically calculate our security metrics from a Java bytecode program. Figure 10.10 shows how the tool is intended to be used, as an analyser at the bytecode level.

The JBSA can be used with bytecode files which were produced from (1) a program created using our SCPA tool i.e. a class security-critical diagram created, annotated Java source code generated and then compiled using the tool; (2) an externally created annotated Java source code program imported into the tool and then compiled using the tool; or (3) an externally created Java bytecode program imported into the tool. In case 3, the bytecode must contain annotations.

The JBSA User Interface is shown in Figure 10.11. To interpret the bytecode instructions, we use a plugin tool for Eclipse from the ASM project called Bytecode Outline [159]. Our tool also uses ASM Java library [159], which is responsible for parsing Java bytecode classes to help us with our analysis.

The program designer will have decided which attribute are classified. These attributes will be annotated as such. One major feature of the JBSA is that it automatically determines which unlabelled attributes may derive their values from classified ones. For instance, if a variable \( x \) is labelled as classified, and the program contains an assignment of the form ‘\( y = E(x) \)’, where \( E(x) \) is some expression involving variable \( x \), then variable \( y \) will be considered by the JBSA to be classified as well. Such a transfer of classified data could also occur via parameter passing.

Annotations: In order to start using the Java Bytecode Security Analyzer, the first requirement is to annotate the Java source code files before compiling them. (Conceptually, we could avoid the compilation step and begin with a bytecode program, given a way of identifying the classified attributes it contains.) The annotation process involves choosing which of the program’s attributes (fields) need to be kept secret. In other words, confidential data is required to be annotated by the programmer/designer as “Classified”. This annotation process can be done using Java 1.6 which allows markers as shown in Figure 10.12, where string ID is marked as classified data.
Another way of identifying which unlabelled attributes may derive their values from classified ones is through control-flow analysis. For example, if a variable $x$ is labelled as classified, and the program contains a statement of the form ‘if ($x == 1$) then $y = 0$;’, then assigned variable $y$ will be considered by the tool to be classified since its value depends on the value of $x$ which is classified.
### Figure 10.13: JBSA Analysis Process

**Analysis Procedure:** The JBSA tool analyses Java bytecode in four distinct steps as shown in Figure 10.13.

1. **Step One—Annotated Classified Data Identification:** The tool first identifies all classes in the analysed program, and all attributes which are annotated as “Classified”, and then marks the enclosing classes as “Critical”. It then looks for two types of annotated methods, those annotated “Classified” and those annotated “Classified-Input”. This represents methods which the programmer thinks may read or access classified data from outside sources such as files or console input using methods like the `read` method in the `java.io.FileReader` class. In this case, the tool examines the method and checks if there is a call to a method which belongs to a Java input package. If so, it marks those attributes whose values are influenced by this call “Classified” and their classes “Critical”.

2. **Step Two—Class Analysis:** This step involves three main activities. First is the identification of private inner/nested classes in the program. The tool checks which classes are private by checking their access flags. Second is to check whether each class implements `java.io.Serializable`, and if so the tool notes this class as a ‘serialised class’. The third activity involves examining each instruction of the code to check whether there is an `invoke` instruction calling a method in the `java.lang.reflect` library. In this case, the tool records this, and hence sets the reflection metric to 1. To avoid the need to parse the program more than once, the tool
also notes which classes are finalised and which are superclasses in this step.

3. Step Three-Information Flow Analysis: In this step, the tool performs a full data flow analysis to confirm the consistency of the programmers’ annotations. This is done to identify further unlabelled attributes which derive their values from classified ones. This involves analysing the flow of information explicitly and implicitly, inspecting each method to identify any flow of data from classified attributes to unclassified ones, either via local variables, from other methods which return classified attributes, or implicitly through control flow statements such as while loops and if-else statements. It then marks every class which contains data identified as classified by this analysis as ‘critical’. This analysis produces the inputs for the attributes’ accessibility, composition, classes’ extensibility and design size-based metrics.

Once classified attributes are identified, the tool records each method as either a mutator, accessor, writer, or some combination of the three based on its instructions. If the method has at least one put instruction involving a classified attribute then this method is considered a ‘classified mutator’. A method is noted as a ‘classified accessor’ if it has at least one get instruction for a classified attribute and as a ‘classified writer’ if it has at least one invoke instruction from a writing class in the java.io library using a classified attribute. (The tool cannot, however, know whether or not a method in another package, not available for analysis also performs input/output operations.) If the method interacts with any classified attribute in any way, the tool records this method as a ‘classified’ one. Next the tool checks if this classified method’s access flag is private. If not, it adds this method’s name and descriptor to a list of the non-private classified methods.

During this analysis, the tool checks how information flows between methods based on how one method calls another. It also checks if a method accesses an attribute indirectly through other methods (as accessing attributes directly has already been checked). Such a method could be another method in the same class, coupled to a method in another class, or calling a method higher in the inheritance hierarchy. If the called method is within the same class, then the tool analyses the method to determine whether it is a classified mutator, classified accessor, classified writer, or any combination of these. If the called method is in another class, it checks the same characteristics as if it is in the class it is called from but it counts the coupling links depending on how many classified
attributes the called method mutates or accesses. If the called method is in the calling method’s class inheritance hierarchy then the tool checks whether the inherited method is a classified mutator, classified accessor, classified writer, and/or any of these but it does not count this as coupling with the classified attribute. This analysis provides the inputs to the calculation of classified methods’ accessibility and extensibility, cohesion, coupling, inheritance-based security metrics.

With regard to the type inference rules in Chapter 9, our tool implements the rules as follows:

- Rules bi-operator and uni-operator require an expression to be labelled classified (high security) if any of its operands is classified. Therefore, our tool checks if the operand of any of these operators has either a get or invoke instruction involving classified data.

- The new rules (i.e., new-par and new-def) require us to label new expressions as classified if they are used to construct an object using one or more classified parameters. Thus, our tool looks for instructions which return values (e.g., get and invoke) and checks if these are used as a parameter to an object’s constructor.

- The cast rule aims to label a variable as classified if the type it is cast to is classified. To achieve this aim, our tool checks for get instructions for variables that are the target of cast instructions. In a case where the new type is classified then this variable is marked as classified.

- The invoke rules (i.e., invoke-def, invoke-par, invoke-par-val and invoke-par-ref) require an expression to be labelled classified if it invokes a method where one of its parameters is classified, if it invokes a method of a classified object or if the invoked method’s return value is derived to be classified. To achieve this, our tool checks if any of the invoke bytecode instructions is preceded by a get instruction over a classified variable or the invoked method has a return instruction involving classified variables.

- The field rule aims to label an expression classified if it has a field that has been marked or derived to be classified. Therefore, our tool checks if there is a get instruction involving a classified field in the same class, classified object, or involving an object’s field that is marked as classified in its class.
10.2. Security-Critical Program Analyser (SCPA) Architecture

- The `const` rule aims to label an expression classified if it has a constant which has been labelled classified. Since Java bytecode do not record symbolic relationships between literals, constants are not considered as having high security in themselves and should not be labelled as classified [117]. Thus, if a constant is labelled classified, then the tool will count these attributes as part of the classified ones but will not be able to detect their flow to other attributes.

- The control flow rules (i.e., `if`, `if-else` and `while`) aim to detect implicit flows of classified information which could arise from control flow statements such as choice and selection statements where a high-security variable influences the execution of assignments to low-security variables. They require us to label a statement controlled by a condition as classified if the condition is derived to be classified. Thus, our tool checks for `if` instructions that are preceded by a classified `get` or `invoke` instruction to mark statements controlled by such conditions as classified.

- The assignment statement rules (i.e., `f-assign`, `ifs-assign` and `sfs-assign`) aim to mark target variables as classified if the expression assigned to them is derived as classified. Therefore, our tool checks if the expression is derived to be classified, and then marks the variable, whether it is a “whole” object or a basic type, as classified. In cases where the variable is assigned in a classified control flow environment, i.e., where the execution of the assignment is conditional based on a high-security expression, the tool marks this variable classified as well. In the case where the variable is an individual local variable, then it is treated as a single-assignment, which means that it is treated differently every time depending on the type of the expression. In situations where the variable is an object and contains field selection, e.g., `obj.field`, then there are several cases:

  (a) If the selected field is a static field, then the field is marked as classified in the class.

  (b) If the field is not static and is already marked as classified in the class, then nothing changes.

  (c) If the field is not static and is unclassified, then the object becomes classified.

4. Step Four-Security Metrics Calculations: Once the previous steps are completed, the tool uses the recorded information to calculate each of the security metrics. The Data
Encapsulation and Cohesion-based security metrics are calculated for each class as well as for the entire program. This process also involves writing all of the classified attributes as well as each class’s classified mutators and accessors to a text file. This file also contains each class’s Data Encapsulation and Cohesion-based security metrics. At the end of the file, a summary of each of the security metrics is written. This file can be generated either in a human-readable form or in a format suitable for input to a spreadsheet program. These results are also displayed and graphed in the JBSA interface (Figure 10.11).

**JBSA Program Code Security Metrics Calculations**  This section shows how our Java Bytecode Security Analyser (JBSA) works on two small examples to extract our security metrics for object-oriented programs. The tool relies on a number of Java bytecode instructions, described in Table 10.1[160]. Each object-oriented property is illustrated using a sample Java source code fragment and its interpretation to Java bytecode in order to explain how the tool calculates each metric.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getfield</td>
<td>Fetch field from object.</td>
</tr>
<tr>
<td>getstatic</td>
<td>Get static field from class.</td>
</tr>
<tr>
<td>putfield</td>
<td>Set field in object.</td>
</tr>
<tr>
<td>putstatic</td>
<td>Set static field in class.</td>
</tr>
<tr>
<td>invokevirtual</td>
<td>Invoke a method based on the class of the object.</td>
</tr>
<tr>
<td>invokespecial</td>
<td>Invoke instance initialisation methods, private methods and superclass methods.</td>
</tr>
<tr>
<td>load</td>
<td>Load the value of a local variable to stack.</td>
</tr>
<tr>
<td>return</td>
<td>Return a value from a method.</td>
</tr>
<tr>
<td>store</td>
<td>Store a certain value into a local variable.</td>
</tr>
<tr>
<td>label</td>
<td>Show the address of a set of bytecode instructions.</td>
</tr>
<tr>
<td>if</td>
<td>Jump to an address depending on the if statement conditions.</td>
</tr>
<tr>
<td>goto</td>
<td>Jump to a certain address.</td>
</tr>
<tr>
<td>pop</td>
<td>Load the top operand value of the stack.</td>
</tr>
<tr>
<td>tableswitch</td>
<td>A jump table for certain addresses depending on their index in the table.</td>
</tr>
</tbody>
</table>

- **Calculating Data Encapsulation-Based Security Metrics:**
To illustrate how the JBSA tool calculates the data encapsulation-based security metrics, consider the Java source code program in Figure 10.14 and an extract of its corresponding bytecode in Figure 10.15. The package is called Details and has two classes, Name and Password. The tool has to first find the annotated classified attributes in the Java bytecode, such as those marked with “Classified” in Figure 10.15. Then it checks which methods are classified, i.e. those which have put and get instructions operating on classified attributes, and recursively checks those methods which have invoke instructions for other classified methods such as the methods in Figure 10.15.

Therefore, with regard to the program in Figure 10.14, the tool will count each non-private classified instance and class attribute (i.e. the lastName and DOB attributes) compared to the overall number of classified attributes (i.e. the lastName, DOB and password attributes from both classes). Similarly, it will count the non-private classified methods which mutate or access classified attributes (i.e. SetName, GetNameDetails and GetPassword) compared to the total number of classified methods (i.e. SetName, GetNameDetails, GeneratePassword and GetPassword). Then, it will check...
whether any of the class’s methods has called another method from the reflection package, which is not so in this case.

After we load the Java bytecode from Figure 10.14 into the JBSA tool, we obtain the security metrics shown in Table 10.2. The CIDA and COA metrics’ values tell us that the Password class has fewer accessible classified attributes and methods than those in the Name class.

- **Calculating Cohesion-Based Security Metrics:**

To calculate the cohesion-based security metrics, the JBSA tool determines which methods have put instructions operating on classified attributes (like GeneratePassword from Figure 10.15b) or those methods which have invoke instructions for classified mutators (like the GetPassword method from Figure 10.15b). Similarly, the tool does the same thing when analysing the accessor classified methods but instead it looks for get instructions as in method GetPassword in Figure 10.15b. Some methods could be classified mutators and accessors at the same time, such as method GetPassword in Figure 10.15b but these methods are counted only once when calculating the CAIW metric. Then, it checks if any of these classified methods write
Table 10.2: Details Package Data Encapsulation and Cohesion Security Metrics

<table>
<thead>
<tr>
<th>Program Unit</th>
<th>CIDA</th>
<th>CCDA</th>
<th>COA</th>
<th>RPB</th>
<th>CMAI</th>
<th>CAAI</th>
<th>CAIW</th>
<th>CMW</th>
<th>CWMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name Class</td>
<td>0.67</td>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
<td>0.5</td>
<td>0.71</td>
<td>0.67</td>
<td>0.50</td>
</tr>
<tr>
<td>Password Class</td>
<td>0.0</td>
<td>0.0</td>
<td>0.50</td>
<td>0.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>0.67</td>
<td>0.0</td>
</tr>
<tr>
<td>Details Package</td>
<td>0.50</td>
<td>0.0</td>
<td>0.75</td>
<td>0.0</td>
<td>0.42</td>
<td>0.34</td>
<td>0.78</td>
<td>0.67</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Table 10.3: Details Package Coupling, Composition, Design Size and Extensibility Security Metrics

<table>
<thead>
<tr>
<th>Program Unit</th>
<th>CCC</th>
<th>CPCC</th>
<th>CDP</th>
<th>CSCP</th>
<th>CCE</th>
<th>CME</th>
<th>UACA</th>
<th>UCAM</th>
<th>UCAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Details Package</td>
<td>0.25</td>
<td>1.0</td>
<td>1.0</td>
<td>0.50</td>
<td>0.50</td>
<td>0.25</td>
<td>0.0</td>
<td>0.75</td>
<td>0.5</td>
</tr>
</tbody>
</table>

classified attributes to the public such as method `GetNameDetails`.

Table 10.2 shows the results for the program in Figure 10.14 once it is compiled and analysed by the JBSA tool. The CAAI and CAIW metrics’ values tell us that the `Name` class has fewer methods which interact with classified attributes than those in the `Password` class, and hence is more secure in this regard. Moreover, the CWMP metric’s values tell us that the `Name` class has more methods which write classified attributes than those in the `Password` class, and hence is less secure in this regard.

- **Calculating Coupling-Based Security Metrics:**

To measure the security-related couplings between classes and classified attributes based on Java bytecode, the JBSA tool acts in a similar way to the cohesion metrics process except that the owner of a relevant instruction is not in the current class. If this other class happens to be within the analysed program, the tool checks whether the called attribute is classified or not. If it is classified then the tool counts this as a link to a classified attribute.

The coupling-based security metric (CCC) for the program in Figure 10.14 is shown in Table 10.3. It reveals that a quarter of the classified attributes in the `Details` package are coupled to another class.

- **Calculating Composition-Based Security Metrics:**
To calculate the composition-based security metrics, the JBSA tool invokes a method in the ASM parsing library which lists all the inner classes for each class in the Java bytecode program. The tool checks if those classes are critical and not private by checking if the class’s access flag is not equal to `Opcodes.ACC_PRIVATE`. Then it calculates the proportion of non-private composed-part critical classes to the total number of critical classes in the program.

The composition-based security metric (CPCC) for the program in Figure 10.14 is shown in Table 10.3, which in this case tells us that there are no private composed-part critical classes in this small program fragment.

- **Calculating Extensibility-Based Security Metrics:**

To obtain the results of the extensibility-based security metrics, the tool looks at the access flags for each critical class and classified method. The ASM parsing library gives a unique access flag (i.e., `Opcodes.ACC_FINAL`) for those classes and methods which are declared in their definitions as `final`. Hence, the tool checks those classes which are critical and do not have this access flag set. Similarly it does the same thing with methods.

The extensibility security metrics results of the Details package (CCE, CME, UACA, UCAM and UCAC) are shown in Table 10.3 and reveal that only half of the critical classes in the Details package are non-extensible and a quarter of the classified methods are non-extensible. The metrics also reveal that all of the assigned classified attributes are accessed and three quarters of the classified accessors were never called by other methods. They also show that half of the critical classes are never used by other classes in the program. Overall, therefore, the package has many vulnerable points at which additional code can access classified data.

- **Calculating Design Size-Based Security Metrics:**

In order to calculate the design size-based security metric, the JBSA tool counts the number of critical classes compared to the total number of classes in the program. Critical classes are those which contain classified attributes annotated as “Classified” or which have attributes which derive their values from critical classes such as the password attribute in the Name class which is of type `Password`. The tool checks for whether any of these critical classes have an implementation for the Serializable Java interface.
The result of these metrics (CDP and CSCP) for the program shown in Figure 10.14 are shown in Table 10.3. These results tell us that all classes in the Details package are critical and half of the critical classes have implemented the Serializable Java interface.

**Calculating Inheritance-Based Security Metrics:**

```java
public class Staff {
    @Marker(" Classified")
    protected String name;
    @Marker(" Classified")
    protected String dob;
    public void setStaff(String _name, String _dob) {
        name = _name;
        DOB = _dob;
    }
    public String getStaff() {
        return name + DOB;
    }

    public class Part_Time_Staff extends Staff{
        private String workingHours;
        public void setHours(String _hours) {
            workingHours = _hours;
        }
        public String getHours() {
            return workingHours;
        }
    }

    public class Full_Time_Staff extends Staff{
        protected double increaseRate;
        public void setRate(double _rate) {
            increaseRate = _rate;
        }
        public double getRate() {
            return increaseRate;
        }

        public class Manager extends Full_Time_Staff{
            @Marker(" Classified")
            protected String accessCode;
            public void setAccess(String _access) {
                accessCode = _access;
            }
            public String getAccess() {
                return accessCode;
            }
        }
    }
}
```

**Figure 10.16: Employee Package Inheritance Hierarchy**

To illustrate the Inheritance-based security metrics, we show how they are produced by the JBSA tool using the Employee package in Figure 10.16. With regard to these metrics, the tool first checks each class to see whether it is a superclass (the Staff class), a subclass (the Manager and Part_Time_Staff classes), or both (the Full_Time_Staff class).

The tool checks the bytecode instructions for each critical class, classified attribute, and classified method. To determine which classes are critical and super, in order to calculate the critical superclass inheritance metrics (CSP and CSI), the tool uses a method from the
ASM library which returns the superclass name of a given class. Then, it checks if this name is within the current files, and if so checks whether it is critical or not. Once the tool has finished this process, it checks which of these superclasses have classified attributes and their access flag is not private (i.e., does not equal `Opcodes.ACC_PRIVATE`). It also does the same thing with regard to classified methods in order to calculate the metrics for inheritable classified attributes and methods (CMI and CAI).

The inheritance-based security metrics for the program in Figure [10.16] produced by the tool are shown in Table [10.4]. They tell us that half of the critical classes are super, half of the classes in the hierarchy interact with critical superclasses, half of the classified methods are in super critical classes, and two thirds of the classified attributes are in super critical classes. Overall, therefore, this design places classified data high in the class hierarchy, where it is more vulnerable.

<table>
<thead>
<tr>
<th>Model</th>
<th>CSP</th>
<th>CSI</th>
<th>CMI</th>
<th>CAI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee</td>
<td>0.50</td>
<td>0.50</td>
<td>0.50</td>
<td>0.67</td>
</tr>
</tbody>
</table>

### 10.3 SCPA Case Study

Having seen how the JBSA does its calculations, the following case study illustrates how our tool can be used in practice to assess a specific security-critical program with regard to our security design and code metrics during the software development process. The case study shows how design and code metrics might vary and how they change when a number of standard refactoring rules are applied to the original program.

In order to accurately allow the UMLDSA to apply the security design metrics on a given design, a complete annotated UML class diagram of that design is required. The class diagram must include UMLsec and SPARK’s annotations in addition to the standard elements of a class diagram. We use our SCPA tool to design two versions of a specific security-critical program such that the second design is a refactored version of the first one using a number of standard refactoring rules which aim to improve security as described in Chapters 5 and 8. We also use the SCPA tool to convert both of these designs to their equivalent Java source code so the JBSA can assess the security of these program based on our security code metrics.
10.3.1 Original Annotated Design

Figure 10.17 shows screen shot of an annotated class diagram designed using our UMLDSA for a planned computer program for storing records of legal cases. It consists of several classes responsible for storing details about a judge, client, court and case. The Judge class is responsible for storing a judge’s information. This includes the judge’s level and title, which are always used together to identify a certain judge. We assume those attributes are confidential data, and hence needs to be kept secret. The Court class contains information related to a certain court, which consists of the court’s name and judges belonging to it. The judges at a certain court are assumed to be security-critical, and therefore they are labelled classified (i.e., “secrecy” in UMLsec).

The Client class is responsible for storing information about clients’ names and their IDs, and we assume that a client’s ID holds confidential data and thus needs to be protected. A client can be either a person or company. In case that the client is a person, then their national
identity number represents their ID in the Client class which is security-critical information. However, when the client is a company then their business number represents their ID, which is not security-critical data. However, the Client class does not distinguish the difference between these two types of IDs. This means that ID might hold classified information when it belongs to a person and therefore this attribute has be to annotated classified and kept secret. The main class of this program is the Case class which stores information about a given case’s type, court and client, which are security-critical data.

10.3.2 Refactored Design

Figure 10.18 shows a screen shot of a refactored version of the original design using a number of standard refactoring rules [18]. The refactoring rules used to get this version aim to make the original program’s design clearer and more maintainable. Furthermore, they aim to improve the design’s overall security as shown in Chapters 5 and 8. This version is designed using UMLDSA and then converted to Java source code using JSCG in order to show how the tool assesses each program differently with regard to our security metrics.

The class diagram in Figure 10.18 differs from the original design in the number of classes, which is now six instead of four. This was done to improve the extendibility and effectiveness of the program [56], and uses the refactoring rules of Extract Superclass to extract a non-critical superclass (Client) and Extract Subclass to extract a critical subclass (Person) and a non-critical subclass (Company). The shared non-classified attribute (name) and non-classified methods (SetName and GetName) for both of the new subclasses were pulled up to their non-critical superclass using the Pull Up Field and Pull Up Method refactoring rules. The fields and methods of these two new subclasses were pushed down to their relevant subclass using the Push Down Field and Push Down Method.

The critical Judge class in this design is composed-part to the Court class and its accessibility is marked as private. This was done using the refactoring rule Extract Composed-Part Class. The Judge class now has fewer classified attributes and methods. This was done using the Inline Field and Inline Method rules to combine the level and title attributes into one attribute ID and combine SetLevel and SetTitle methods into one classified method i.e., SetID. This was performed since these attributes and methods were always used together to identify a certain judge.
This design also differs from the original as the accessibility of its attributes has changed to be private using the refactoring rule of Encapsulate Field except for the name attribute whose accessibility changed to be protected since it is used by the subclasses Person and Company. The refactoring rules Hide Method was also used to change the accessibility of SetJudgeID and GetJudgeID methods in the Court class since they are only used within their class.

10.3.3 Security Design-Metrics Results

The two tables in Figures 10.17 and 10.18 show the security design metrics results produced by our UMLDSA. They show the design’s security characteristics (i.e., absolute metrics) for the analysed designs and their annotated security-critical attributes, methods, and classes. It can be seen that Design 2 has more methods and classes than Design 1, but fewer classified attributes and methods.
They also show our security design metrics calculated based on the two designs. The results of these metrics show that all values of Design 2 are lower than those in Design 1. This, in fact, reflects the general characteristics of these designs in which the absolute number of classified attributes and methods are lower for Design 2 than those for Design 1.

Given that lower values of each metric are considered more secure, it can be seen that the results of these two different designs vary with regard to their security level for many of the metrics. Design 2 has the lowest values for all of the security design metrics, and hence it is the most secure design in this regard.

10.3.4 Security Code Metrics Results

Our security code metrics measure security based on a given object-oriented program’s executable code (Java bytecode in our case) with respect to the (potential) flow of ‘classified’ data. Using this approach we aim to capture the exact behaviour of the program at the execution
The two tables in Figures 10.19 and 10.20 show the security code metrics results produced by our JBSA based on the executable code of the two designs. It can be seen from these two tables that the characteristics of these two programs are similar to the ones produced by the UMLDSA with regard to their designs. They differ in the number of methods and classified methods. This is because bytecode classes have default constructors for each class, each of which is counted as a method. The increase in the number of classified methods in Program 1 compared to its design (Design 1) is caused by SetCourtName and SetClientName. Although these methods have not been annotated as ‘classified’, JBSA marks them ‘classified’ since they interact with an object that has been labelled as ‘classified’. This is similar to the increase in the number of classified methods in Program 2, in which SetCourtName is marked ‘classified’ since it interacts with a classified object (i.e., court).

The other difference between the results of program and design absolute metrics is the
number of attributes in Program 2 when compared to its design. While the number of attributes shown in the results of Program 2 is 10, its design result has only shown that Design 2 has 9 attributes. The cause of this difference is that compilers by default generate an attribute that represents a composed-part class in its composed-whole class when these classes are compiled. Since Program 2 has one composed-part class (i.e., Judge), an object called Court$judge is generated by default in the compiled Court class. In general, it can be seen that Program 2 has more methods and classes than Program 1, but fewer classified attributes and methods, which is a similar case to the designs of these programs.

With regard to the results of our security code metrics shown, it is obvious that all results of these metrics have either stayed the same or decreased, except for two metrics (UCAM and UCAC) which have slightly increased. This indicates that Program 2 is more secure than Program 1, which is what we have identified based on the security design metrics of the designs of these programs.

In summary, the tool-supported implementation of our security code metrics have shown hidden attributes and methods which cannot be seen at the design level. These include the default generated constructors for each class and the default generated attribute for each composed class. While security design metrics can measure security in an easy way, security code metrics can give more accurate results.

### 10.3.5 Security Metrics Analysis

Our JBSA can produce the hierarchical security metrics defined in Chapter 7, which presents four different levels of interpreting the results of our security code metrics as shown in Figure 10.11. One way is by showing the security data flow metrics which are shown in Figures 10.19 and 10.20. Another way is by showing the readability and writability metrics level. This can tell us which of the programs is more secure in terms of readability and writability of security-critical data. Figures 10.19 and 10.20 show these metrics and indicate that Program 2 is more secure than Program 1 with regard to the readability and writability of classified data. This means that Program 2 reads and writes fewer classified attributes and methods, and critical classes than Program 1.

Another way of interpreting the results of our metrics is to choose which program satisfies the requirement of specific design principles. This is the third level on our hierarchical security
metrics defined in Chapter [7]. The radar chart in Figure [10.11] shows the values of each of these programs with regard to seven different security design principles. It can be seen that Program 2 has the lowest values of all of these security design principles, and hence it meets the requirements of all of them. Therefore, we can conclude that Program 2 is the most secure program in this regard.

The easiest way of identifying the most secure program is by comparing the Total Security Index (TSI) for each of the programs. The TSI values for these programs, which is produced by our JBSA, also confirms our judgement that Program 2 is the more secure program. Given that lower values of TSI are considered more secure, our JBSA shows that the TSI value for Program 1 is 74.655 while the TSI value for Program 2 is 53.301.

The other way of interpreting the results of these metrics is to identify how the chosen refactoring rules have affected security. We have chosen a number of standard refactoring rules which were shown in, Chapters [5] and [8] to improve security. The results shown for both of these programs confirm the goal of the chosen refactoring rules as security has improved when Design 1 has been refactored using these rules to Design 2.

10.4 Conclusion

In this chapter, we have shown how our Security-Critical Program Analyser (SCPA) automatically assesses the overall security of security-critical designs and programs. Our tool allows developers to generate an annotated UML-like design for their security-critical programs and quantify its overall security with regard to our security design metrics. We have also shown how our tool generates Java source code for an existing security-critical design. This can be also compiled using our tool, in which case the SCPA uses an external compiler, to generate its Java bytecode program. In order to produce accurate results, SCPA assesses the security of a given program with regard to our security program metrics based on its annotated bytecode classes.
Part V

CONCLUSION
Conclusions and Recommendations for Future Work

This chapter summarises the main outcomes which this thesis has achieved in order to fully solve the defined research problem. Finally, it describes several directions for possible future work related to the area of research described in this thesis.

11.1 Summary of the Research

Software metrics promise an easy way of comparing the relative security of programs or assessing the security impact of code modifications. The main problem addressed by this thesis was to find an easy approach for evaluating the security level of a given program at various stages of its development life cycle. We have solved this problem by identifying a set of security metrics that allow us to easily measure the relative security of object-oriented programs during the design, coding, and refactoring (maintenance) stages of software development. We have interpreted ‘security’ to mean control over data confidentiality, i.e., the ability to read classified values, and data integrity, i.e., the ability to update classified variables. Both of these properties concern the (potential) flow of classified data, and therefore our security metrics measure security from this point of view.
Chapter 11. Conclusions and Recommendations for Future Work

11.1.1 Security Design Metrics

The first outcome of this thesis is a set of security metrics which assess program security with regard to its design. 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 [28] and SPARK’s annotations to express the information flow relations between attributes, methods and classes [17]. These metrics allow designers to compare the security of various alternative designs for a given object-oriented program, by quantifying potential information flow from ‘classified’ data values. Our design-level security metrics measure security from the perspective of information flow based on the security design principles of “reducing the size of the attack surface” [40] and “least privilege” [33, 32], based on security annotations introduced by the designer. They are derived from seven different object-oriented quality properties: data encapsulation, cohesion, composition, coupling, extensibility, inheritance and design size as an adjunct to other well-established metrics for assessing design or program complexity [56]. Each metric is a ratio in the range 0 to 1, with lower values considered more secure.

11.1.2 Security Code Metrics

Another outcome of this thesis is another set of security metrics that assess security of object-oriented programs with regard to their source code. Our security code metrics are measurable at the level of bytecode instructions so we can capture the exact behaviour of a Java program in the Java Virtual Machine, which gives accurate results. Similar to our security design metrics, these security code metrics assess the security of a given object-oriented program with regard to the way its object-oriented design properties (data encapsulation, cohesion, composition, coupling, extensibility, inheritance and design size) influence the accessibility of any classified data it contains. They aim to reveal many of the vulnerabilities associated with insecure code. To validate the metrics, we have used our software tool to analyse the Java bytecode of a number of large-scale Java projects in order to quantify their security.
11.1.3 Security Hierarchical Model

Another main outcome of this thesis is a hierarchical security assessment model which provides an approach for measuring security of object-oriented programs at various levels of abstraction. The model builds on the characteristics of our code level metrics in order to identify another set of security metrics that measure higher-level characteristics such as the readability and writability of classified attributes. These new characteristics are then grouped around security design principles in order to identify another set of security metrics. The top of the hierarchy is a single value that measures security based on the values of all the lower-level metrics.

11.1.4 Secure Refactoring Rules

Another outcome of this thesis focuses on using our metrics to determine how to refactor programs in a way that enhances their overall security. Refactoring has been used widely to improve the reusability, maintainability and performance of programs, but the effect of refactoring on security has received little attention in previous studies. In this thesis, we have studied the impact of a number of standard design and code-level refactoring rules on security in order to identify which rules improve or worsen security. We have used our security metrics to compare the security of different refactored versions of the same design or program, which thus allowed us to determine the impact of the refactoring rules on security. These results can be used as a guidance for maintaining security-critical programs and as an alert for potentially unsafe maintenance steps.

11.1.5 Security Assessment Tool Support

The final outcomes of this thesis is a set of type inference rules for defining valid information flow and a tool that automatically assesses security with regard to such flow in order to calculate our security metrics. Our type inference rules formally define the capabilities of our security assessment tool which is called the “Security-Critical Program Analyser (SCPA)”. SCPA consists of a tool chain with four sub tools each responsible for accomplishing a certain task. The first is a UML Design Security Analyser (UMLDSA) for allowing software designers to design their security-critical programs in a UML-like class diagram and annotate these designs
Chapter 11. Conclusions and Recommendations for Future Work

with UMLsec and SPARK’s annotations. UMLDSA then allows users to assess their security-critical designs with regard to our security design metrics. A Java Source Code Generator (JSCG) is the second sub tool of SCPA, and is responsible for producing Java source code programs. It takes the output of the UMLDSA which is a security-critical design in order to produce an equivalent Java program skeleton. The third sub tool is an off-the-shelf external Java compiler which is used to compile JSCG generated Java programs or separately-produced Java program. It produces Java bytecode classes that can be used by the fourth sub tool of SCPA which is a Java Bytecode Security Analyser (JBSA). JBSA is responsible for analysing Java bytecode classes to assess the program’s relative security with regard to our security code metrics. JBSA traces the security levels of the program’s variables based on our type inference rules, and produces a hierarchical set of security metrics.

11.2 Recommendations for Future Work

A number of long-term future work recommendations can arise from this thesis. One important recommendation is concerned with providing another form of validation of our security metrics including user trials with security-critical software engineering experts in order to independently validate the metrics. Another is related to considering subtle forms of information flow and further development of the type inference system to cover other program constructs such as exception handling. Another is related to studying the impact of individual refactoring rules in combination. However, the following items describe in detail further recommendations for future work arising directly from the outcomes of this thesis.

11.2.1 Adding Non-Security-Critical Code

One obvious limitation associated with some of our security metrics is that we can make programs seem “more secure” simply by adding non-security-critical code to them (or vice versa) i.e., by decreasing the proportion of security-critical code. These metrics are related to the properties of cohesion, coupling, design size and the inheritance CSI metric. On the other hand, other metrics related to data encapsulation, composition, extensibility and the inheritance metrics (CSP, CMI and CAI) do not have this problem. However, this problem is an inevitable one for any metric which is the ratio of two values. In fact, software metrics in general have been
11.2. Recommendations for Future Work

criticised for such limitations, and our metrics are no exception. For instance, “lines of code” as 
a measure of software size is frequently criticised because it can be changed by reformatting the 
program. Therefore, this seems to set a direction for future work on how present the security 
metrics in a way that makes the programmer aware of this inherent characteristic.

11.2.2 Reconciling Absolute and Relative Measures

Another unresolved issue is related to our hierarchical security metrics model. Our model aims 
to quantify security by combining the results of lower level metrics into high level ones. To 
do this we combine absolute and relative security measurements into the same metric, i.e., the 
total security index (TSI). This attempt to combine two different kinds of metrics sometimes 
produces counter intuitive results.

This is clearly shown by the TSI values in Table 11.1 for the Jacksum program which 
has been analysed in Chapter 6. Its TSI values show a worsening in overall security for its 
successive revisions. This, however, contradicts the impression given by the relative code 
metrics (shown in Chapter 6) that generally show an apparent improvement in each Jacksum 
version. Upon analysis of the program’s characteristics we found that it had gone through a 
major revision which caused it to have numerous new features added. This included adding a 
considerable amount of security-critical code which dramatically increased the security-related 
absolute metrics, and cased a net increase in the program’s total security index. Whether or 
not this increase in the absolute amount of security-critical code should outweigh the evident 
decrease in the relative proportion of security-critical code is debatable. Thus, there may be 
scope for adjusting the weights in Chapter 7 to accommodate different tradeoffs of absolute 
versus relative values. The basic question is which is most important, the absolute size of the 
attack surface or its proportion within the program?

<table>
<thead>
<tr>
<th>Program</th>
<th>Release 1</th>
<th>Release 2</th>
<th>Release 3</th>
<th>Release 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSecurity</td>
<td>57.0111</td>
<td>57.0046</td>
<td>58.964</td>
<td>55.928</td>
</tr>
<tr>
<td>Kasai</td>
<td>42.72</td>
<td>42.716</td>
<td>42.716</td>
<td>42.716</td>
</tr>
<tr>
<td>Jacksum</td>
<td>82.415</td>
<td>97.539</td>
<td>100.586</td>
<td>118.593</td>
</tr>
<tr>
<td>jGuard</td>
<td>77.552</td>
<td>77.552</td>
<td>77.552</td>
<td>77.552</td>
</tr>
<tr>
<td>JXplorer</td>
<td>582.773</td>
<td>664.9</td>
<td>769.536</td>
<td>738.735</td>
</tr>
</tbody>
</table>

Table 11.1: Total Security Indexes for the Programs Analysed in Chapter 6
11.2.3 Multiple Security Levels

We have developed our security metrics based on a binary classification of data, either is classified or unclassified. However, there exist other models with other classifications of information such as the common model used by Defence departments which divides information into four security levels; Top Secret, Secret, Confidential and Unclassified [161]. A possible extension to this thesis is to adopt this model for how our security metrics classify and measure security-critical information flow.

11.2.4 Visualisation Tool for Security Metrics

Another area of possible future work based on our security metrics is to develop a programming environment that allows software developers to visualise the relative security of their programs. Higo et al. [162] developed an approach which proposes identifying buggy modules by looking for code whose metrics have changed a lot between versions. We could adopt such an approach to our security metrics to develop a plug-in tool that allows programmers to identify how classified data in a given program flows between various attributes, methods and classes. This will allow programmers to compare the relative security of different versions of their programs with ease.

11.2.5 Refactoring Tool for Security-Critical Programs

Finally, another direction for future work inspired by this thesis is concerned with automating the refactoring process for security-critical programs. Such a tool would be similar to the one developed by Sison et al. [163], which is an Eclipse plug-in that automatically makes recommendations for software refactorings by examining the structure of the program’s abstract syntax tree (AST). In our case, we could have a similar tool that studies the program’s abstract syntax tree (AST) and information flow of classified data in order to make recommendations for secure refactoring steps that guarantee to produce more secure programs.
Publications

The following papers describing the findings of this thesis have been published (or have been submitted for publication).


Bibliography


