Developing Secure Systems: A Comparative Study of Existing Methodologies

Bandar M. Alshammari, Colin J. Fidge, and Diane Corney

***Abstract*—With the increasing demand for developing high-quality and more reliable systems, the process of developing trustworthy computer software is a challenging one. In this paper, we review various approaches to producing more secure systems. This includes established general principles for designing secure systems. It also provides an introduction to general software quality measurements including existing software security metrics. This paper also includes a comparison of the various security metrics for developing secure systems (i.e., architectural, design, and code-level metrics). Lastly, the paper examines the approach of refactoring, illustrates its objectives, and shows how refactoring is generally used for enhancing the quality of existing programs from the perspective of information security. At the end of this paper, we provide a discussion of these three approaches and how they can be used to provide guidance for future secure software development processes.**

***Index Terms*—Security design principles, object-orientation, security metrics, secure refactoring.**

I. Introduction

Much existing software is designed with poor consideration of information security which makes it vulnerable to many threats including malicious attacks [1]. Software patches are one of the suggested solutions for many of the security attacks facing software [1] 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 [2]. Security processes, in general, consider many aspects of system design, coding, testing, and auditing [2] (e.g., international security standards such as the Common Criteria [3] or the Trusted Computer Criteria [4]).

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 [5]). However, these solutions do not always work effectively and may, in general, even introduce new vulnerabilities to existing software [1]. Adding security features to systems after they have been developed and deployed has been a major cause of many system vulnerabilities [6]. Therefore, applying security principles from the early stages of the software development life cycle (SDLC) would be a better solution [1] and allow a more coherent system to be produced [6].

Developing a secure system requires a good overall design which takes security into account from the beginning. A suggested methodology by Fernandez [1] 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 [7] 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 [7]. Preventing unauthorised access to sensitive data is the main goal of this approach [7].

Although these approaches can be a starting point for developing secure software, they do not provide an approach for quantifying the security of a given program or design. We therefore survey the current literature to show the best approach which can assist systems' designers to develop more secure systems from the point view of information flow of confidential data. 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.

II. 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 [8], [9]. Furthermore, the mechanisms that make a program secure should be self evident in its design. Bishop's [10] 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 [12] in 1975 was one of the first on software security design principles.

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 [10]. It also reduces the interactions of system components which results in fewer security checks [10]. 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 [10]. Although, there are many defined design principles for developing secure systems, in this section we only concentrate on the ones commonly used to

Manuscript received October 10, 2014; revised January 5, 2015.

Bandar M. Alshammari is with Aljouf University, Saudi Arabia (e-mail: bmshammeri@ju.edu.sa).

Colin J. Fidge is with Queensland University of Technology, Australia (e-mail: c.fidge@qut.edu.au).

Diane Corney is with Oracle, Australia (e-mail: diane.corney@oracle.com).

DOI: 10.7763/LNSE.2016.V4.239

achieve such goal.

A. Principle of 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' [8], which minimises what needs to be verified secure [8]. This leads to fewer security-critical functions which mean 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], [13].

Buffer overflow occurs when data is copied to a place in memory that is bigger than the reserved size [13]. Its threat arises from the fact that such a bug can have an effect on the system's data integrity [13]. 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 strca, can be replaced with gets(buf,size,stdin), strncp, and strncat respectively [11].

B. Principle of Least Privilege

Bishop defines this principle as "a subject should be given only those privileges that it needs in order to complete its task" [10]. 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" [12]. 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 [10].

This principle can be described as "programs and users should run with the least privilege to complete their job" [12]. The main advantage of adhering of this is that it can limit damage from implementation errors or attacks [12]. Another advantage is to minimize the interactions among privileged programs [12]. 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 [12].

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 [10]. 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 [10].

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 [14]. A capability based protection system is used to accomplish this by reducing the authority of objects [14].

C. Principle of 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" [10]. Saltzer stated [12] that this principle was suggested by E. Glaser in 1965 so that access decisions are based on permission rather than on exclusion [12]. It also means that the default situation is lack of access and the protection mechanism scheme should identify conditions under which access is permitted [12]. 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 [12]. 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 [12].

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 [10]. 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 [10]. However, if a mail server has failed to create a 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 [10].

D. Principle of Economy of Mechanism

Bishop defines this principle as "security mechanisms should be as simple as possible" [10]. 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 [12]. 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 [12].

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 behavior" [15]. 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].

E. Principle of Least Common Mechanism

This principle is defined as a "mechanism used to access resources should not be shared" [10]. 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 [10]. 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 [12], for instance security concerns. It is also stated that shared functions should be run in users' interfaces rather than as a system procedure [12].

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 [10].

F. Principle of 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 [16]:

- Ensure secure defaults (i.e., deny access).
- Undo all changes and restore to a secure state.
- Always make sure to check values for failure.
- 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 [16]. A possible case is revealing sensitive information after a system fails which helps attackers to establish an attack [16]. Howard and LeBlanc verify that revealing unnecessary information about a system failure could help attackers [2]. The golden rule when failing securely is "to deny by default and allow only once you have verified the conditions to allow" [2]. The following example explains [16] the golden rule stated by Howard and LeBlanc [2].

The code in Fig. 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 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 \sloppy "`ERROR NOT ENOUGH MEMOR`" [16]. In this case, the user will grant access because result `dwRet` is not equal to "`ERROR ACCESS DENIED`" [16]. On the other hand, the first code fragment in Fig. 1 shows that if method `IsAccessAllowed` fails for any reason, no access would be granted [16].

```
// This can lead to a secure failure
   DWORD dwRet = IsAccessAllowed(...);
   if (dwRet == NO_ERROR)
   {
      // Secure check OK.
      // Perform task.
   } else {
      // Security check failed.
      // Inform user that access is denied.
   }
```

```
// This Can cause an insecure failure
   DWORD dwRet = IsAccessAllowed(...);
   if (dwRet == ERROR_ACCESS_DENIED)
   {
      // Security check failed.
      // Inform user that access is denied.
   } else {
      // Security check OK.
   }
```

Fig. 1. Faile securely example.

G. Principle of 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 [8]. This means that those isolated domains will interact only with those domains and environments allowed by the security policy [8]. To achieve such isolation, there are four suggested ways: temporally, physically, cryptographically, or logically [8].

H. Principle of Reduce the Size of the Attack Surface

Howard [17] 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 [17] 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*. [18]–[20]. 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) [17].

III. Software Quality Measurements

Several studies have developed metrics for quantifying software quality attributes of object-oriented applications such as reusability and functionality [21]–[26]. 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 [27] suggested a number of object-oriented metrics. This was followed by Chidamber and Kemerer's work [21], [22] 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 [28]. 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, understandability, testability and maintainability [29].

Briand *et al*. [30] 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*. [31] defined a unified framework for measuring coupling in object-oriented programs.

Bansiya [32] 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) [33]. 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.

A. Software Security Metrics

Many software quality attributes have been studied and measured, including maintainability, performance, reusability, and reliability [33]. 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 [2], [9], [34]–[38]. The National Vulnerability Database of the National Institute of Standards and Technology [39] 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 do not belong to any of the above classification.

Another technique used by Maruyama [35] and Howard and LeBlanc [2] 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 [40] 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 [18] or at the low level of individual code structures [41]. 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*. [42] 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 [42].

Further work in this area was conducted by Liu *et al*. who proposed a model called the "User System Interaction Effect (USIE)" [43]. The USIE model is responsible for providing a systematic approach to identify security defects from the architecture of a service-oriented system [43].

A recent approach for measuring security based on architecture is defined by Manadhata *et al*. [18] which 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 [20]. It is based on the set of possible resources which an attacker could use to attack the system [20], including methods, data and channels [20]. A method is described by Manadhata *et al*. as a system entity which could send data (exit point) or receive data (entry point) [18]. Data in their approach is any entity which is visible in the current system such as files, cookies and database records [18]. 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 [20]. 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 of different versions of two IMAP servers and two open source FTP demons [20].

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 [44], [45]. 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 [46] 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 [46].

Another work in this field was by Agrawal *et al*. [47] who defined a measurement of object-oriented class vulnerabilities. The measurement aims to count the number of vulnerable classes in a given design [47]. A vulnerable class is the one which has sensitive and confidential data members and methods [47]. 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 [48]. 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 [48].

Recent studies conducted by Alshammari *et al*. [49, [50] had defined several security metrics for UML class designs, and described a tool for automatically evaluating such metrics [51]. These metrics aim to assess the potential flow of classified data by measuring the accessibility of such data based on the security design principles of "granting least privilege" [12], [10], and [40] and "reducing the size of the attack surface" [17], [20].

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*. [41] 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 [41]. 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 [41]. 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 [41]. 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*. [52], 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 [52].

Alves-Foss and Barbosa [53] 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 [53]. 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 [53].

A similar method to Alves-Foss and Barbosa's [53] is Alhazemi *et al*'s. [54] 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 [54]. The authors validate their metric on five operating systems consisting of three successive versions of Windows and two versions of Red Hat Linux [54].

Another similar work has been conducted by Alshammari *et al*. [55] which described a number of security metrics for object-oriented programs which are measurable at the level of bytecode instructions. The main objective of this approach is to capture the exact behavior of a Java program in the Java Virtual Machine to give more accurate results. Such metrics will provide developers with a simple way of identifying and fixing security vulnerabilities which might occur from the perspective of information flow of security-critical data.

B. 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. Such security metrics need to be capable of measuring overall security of a given object-oriented program based on many of its compositional properties and information flow principles.

IV. Refactoring

Refactoring is an important aspect of software evolution since it aims to increase the quality of software [56]. 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 behavior" [15]. 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 [56]: 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 [15]. On the other hand, refactoring aims to make the code easier to understand but does not always enhance performance [15].

Using refactoring involves two main activities: 1) adding functions and 2) restructuring those functions [15]. Adding functions involves inserting new tasks without changing the existing code and testing those tasks to ensure that they are implemented correctly [15]. The second activity is to restructure the interactions between these new functions in a better way [15]. 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 [56]. Fowler [15] has listed 72 refactoring rules in detail. In addition, the refactoring web page includes more refactoring rules [57]. However, none of these mention the impact of refactoring on security.

A. Refactoring and Security

Refactoring is an important aspect of software evolution since it aims to increase the quality of software [56]. However, its impact on software security was hardly mentioned or studied until recently [35]. Maruyama and Tokoda [58] 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 may 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 [35]. 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 [35]. Another rule aims to protect the secrecy of confidential fields through the rule of Replace Reference with Copy [35]. 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 [35].

Furthermore, Smith and Thober [59] have identified a refactoring approach for critical systems similar to the approach of Li and Zdancewic [60]. 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 [59].

Another work in the area of secure refactoring is Hafiz's [61] 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 [62] 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 [15] may improve security by hiding fields which contain secret data from public access. Similarly, the refactoring rule of *Encapsulate Class with Factory* by Kerievsky [63] 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.

B. 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.

V. Discussion

Our literature review has focused on three main parts concerned with software security. The first is related to software security design, the second has illustrated some of the existing work on software quality measurements including existing software security metrics, and the third has discussed refactoring and how it affects security. These can be used as guidance for future secure software development processes and as elements of 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. However, most of them are not capable of quantifying the security of a given program either from its design or code level. Furthermore, those ones cannot detect the change to security when programs are refactored. Thus, the most efficient and promising security metrics are those which objectively measure the security of

various programs from either the design or code levels. Such metrics are 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 previously defined metrics [49], [50], [55], and [64] differ from previous work as it focuses on the security of the overall program module structure. We had studied 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 challenge for these metrics is how to validate the outcomes. A study conducted by Tempero [65] 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 [65]. This result is consistent with the author's previous studies concerned with to what extent software design decisions are not followed [66] and also consistent with another study which confirmed that many interfaces in programs are not implemented [67]. Having seen the value of such an empirical approach, it is necessary to analyse suits of open-source software to validate security metrics.

VI. Conclusion

In this paper, we have reviewed the literature to describe the various approaches for developing secure systems. It has been shown that the most promising approaches for achieving such goal consist of security design principles, security quality metrics and secure refactoring. Furthermore, an approach which aims to define security metrics that takes into account security design principles from an early stage of development would be an efficient approach for measuring security. Such metrics will also assist in defining the refactoring rules which either increase or worsen security for any given program with regard to the potential flow of classified information. This will also provide guidance for designing and recognising secure systems.

References

[1] E. B. Fernandez, "A methodology for secure software design," in *Proc. the International Conference on Software Engineering Research and Practice, SERP '04*, vol. 1, Las Vegas, USA, 2004, pp. 130–136.

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

[3] CC Recognition Arrangement. (July 2009). The Common Criteria for Information Technology Security Evaluation (CC). [Online]. Available: http://www.commoncriteriaportal.org/

[4] The National Computer Security Center (NCSC). (1985). The Trusted Computer System Evaluation Criteria (TCSEC). [Online]. Available: http://csrc.nist.gov/publications/history/dod85.pdf

[5] R. C. Seacord, *Secure Coding in C and C++*, Upper Saddle River, NJ: Addison-Wesley, 2006.

[6] M. J. Peterson, J. B. Bowles, and C. M. Eastman, "UMLpac: An approach for integrating security into UML class design," in *Proc. the IEEE Southeast Conference*, J. B. Bowles, ed., 2006, pp. 267–272.

[7] B. F. Eduardo and Y. Xiaohong, "Securing analysis patterns," presented at the Fourty Fifth Annual Southeast Regional Conference, Winston-Salem, North Carolina, ACM, 2007.

[8] W. Young, "Verifiable computer security and hardware: Issues," *Tech. Rep.*, Austin, Texas, September 1991.

[9] OWASP. (2010). The open web application security project. [Online]. Available: http://www.owasp.org

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

[11] J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*, Boston: Addison-Wesley, 2002.

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

[13] M. Dowd, J. McDonald, and J. Schuh, *The Art of Software Security Assessment Identifying and Preventing Software Vulnerabilities*, Addison Wesley Professional, 2006.

[14] A. Spiessens, "Patterns of safe collaboration," PhD thesis, University of Catholique de Louvain, Belgium, 2007.

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

[16] S. Barnum and M. Gegick. (2005). Build security in. [Online]. Available: https://buildsecurityin.us-cert.gov/bsi/home.html

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

[18] P. K. Manadhata, K. M. C. Tan, R. A. Maxion, and J. M. Wing, "An approach to measuring a system's attack surface," *Tech. Rep. CMU-CS- 07-146*, Carnegie Mellon University, Pittsburgh, PA, August 2007.

[19] P. K. Manadhata, "An attack surface metric," PhD thesis, Computer Science Department, Carnegie Mellon University, December 2008.

[20] P. Manadhata and J. Wing, "An attack surface metric," *IEEE Transactions on Software Engineering*, no. 99, p. 1, 2010.

[21] S. R. Chidamber and C. F. Kemerer, "Towards a metrics suite for object oriented design," in *Proc. on Object-Oriented Programming Systems, Languages, and Applications*, New York, NY, USA, pp. 197–211, ACM, 1991.

[22] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, pp. 476–493, 1994.

[23] M. Lorenz and J. Kidd, *Object-Oriented Software Metrics: A Practical Guide*, Englewood Cliffs, NJ: PTR Prentice Hall, 1994.

[24] V. R. Basili, "Gqm approach has evolved to include models," *IEEE Software*, vol. 11, pp. 1–8, 1994.

[25] F. B. E. Abreu, M. Goulão, and R. Esteves, "Toward the design quality evaluation of object-oriented software systems," in *Proc. the 5th International conference on Software Quality*, Austin Texas, 1995.

[26] B. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*, Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[27] K. L. Morris, "Metrics for object-oriented software development environments," Master's thesis, Massachusetts Institute of Technology, Sloan School of Management, 1989.

[28] R. Subramanyam and M. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *IEEE Transactions on Software Engineering*, vol. 29, no. 4, pp. 297–310, 2003.

[29] U. Kulkarni, Y. Kalshetty, and V. G. Arde, "Validation of ck metrics for object oriented design measurement," in *Proc. International Conference on Emerging Trends in Engineering and Technology*, pp. 646–651, 2010.

[30] L. C. Briand, J. W. Daly, and J. K. Wuest, "A unified framework for cohesion measurement," in *Proc. the Fourth International Symposium on Software Metrics*, IEEE Computer Society, 1997.

[31] L. C. Briand, J. W. Daly, and J. K. Wuest, "A unified framework for coupling measurement in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 25, pp. 91–121, 1999.

[32] J. Bansiya, "A Hierarchical Model for Quality Assessment of Object-Oriented Designs," PhD thesis, The University of Alabama in Huntsville, 1997.

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

[34] B. Chess and J. West, *Secure Programming With Static Analysis*, Upper Saddle River, NJ: Addison-Wesley, 2007.

[35] K. Maruyama, "Secure refactoring - improving the security level of existing code," in *Proc. the Second International Conference on Software and Data Technologies*, J. Filipe, B. Shishkov, and M. Helfert, eds., Barcelona, Spain, 2007, pp. 222–229.

[36] J. Viega, G. McGraw, T. Mutdoseh, and E. Felten, "Statically scanning java code: finding security vulnerabilities," *Software, IEEE*, vol. 17, no. 5, pp. 68–77, 2000.

[37] J. Viega, J. Bloch, Y. Kohno, and G. McGraw, "Its4: a static vulnerability scanner for C and C++ code," in *Proc. 16th Annual Conference on Computer Security Applications*, pp. 257–267, Dec. 2000.

[38] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi, "A taxonomy of computer program security flaws," *ACM Comput. Surv*., vol. 26, pp. 211–254, September 1994.

[39] NVD. (August 9, 2010). National vulnerability database. [Online]. Available: http://nvd.nist.gov/

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

[41] I. Chowdhury, B. Chan, and M. Zulkernine, "Security metrics for source code structures," in *Proc. the Fourth International Workshop on Software Engineering for Secure Systems*, Leipzig, Germany, pp. 57–64, ACM, 2008.

[42] P. Antonino, S. Duszynski, C. Jung, and M. Rudolph, "Indicator-based architecture-level security evaluation in a service-oriented environment," in *Proc. the Fourth European Conference on Software Architecture: Companion Volume*, New York, NY, USA, pp. 221–228, ACM, 2010.

[43] Y. Liu, I. Traore, and A. Hoole, "A service-oriented framework for quantitative security analysis of software architectures," *IEEE Asia-Pacific Services Computing Conference*, pp. 1231–1238, 2008.

[44] A. Sachitano, R. O. Chapman, and J. A. Hamilton, "Security in software architecture: A case study," in *Proc. from the Fifth Annual IEEE SMC Information Assurance Workshop*, pp. 370–376, 2004.

[45] E. A. Schneider, "Security architecture-based system design," in *Proc. the 1999 Workshop on New Security Paradigms*, Ontario, Canada, pp. 25–31, ACM, 2000.

[46] S. Chandra and R. A. Khan, "Software security metric identification framework (SSM)," in *Proc. the International Conference on Advances in Computing, Communication and Control*, New York, NY, USA, pp. 725–731, ACM, 2009.

[47] A. Agrawal, S. Chandra, and R. Khan, "An efficient measurement of object oriented design vulnerability," in *Proc. International Conference on Availability, Reliability and Security*, pp. 618–623, 2009.

[48] A. Agrawal and R. A. Khan, "Impact of inheritance on vulnerability propagation at design phase," *SIGSOFT Softw. Eng. Notes*, vol. 34, pp. 1–5, July 2009.

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

[50] B. Alshammari, C. J. Fidge, and D. Corney, "Security metrics for object-oriented designs," in *Proc. the Twenty-First Australian Software Engineering Conference*, J. Noble and C. J. Fidge, eds., California, USA, pp. 55–64, IEEE Computer Society, 2010.

[51] B. Alshammari, C. Fidge, and D. Corney, "An automated tool for assessing security-critical designs and programs," in *Proc. the National Workshop on Information Assurance Research*, Riyadh, Saudi Arabia, April 18, pp. 1–10, VDE, 2012.

[52] K. Aggarwal, Y. Singh, A. Kaur, and R. Malhotra, "Software design metrics for object-oriented software," *Journal of Object Technology*, vol. 6, pp. 121–138, January 2006.

[53] J. Alves-Foss and S. Barbosa, "Assessing computer security vulnerability," *SIGOPS Oper. Syst. Rev*., vol. 29, no. 3, pp. 3–13, 1995.

[54] O. Alhazmi, Y. Malaiya, and I. Ray, "Measuring, analyzing and predicting security vulnerabilities in software systems," *Computers & Security*, vol. 26, no. 3, pp. 219–228, 2007

[55] B. Alshammari, C. J. Fidge, and D. Corney, "Security metrics for java bytecode programs," presented at the Twenty-Fifth International Conference on Software Engineering and Knowledge Engineering, Boston, Knowledge Systems Institute, June 27–29, 2013.

[56] J. Garzas and M. Piattini, *Object-Oriented Design Knowledge: Principles, Heuristics, and Best Practices*, Hershey, PA: Idea Group Pub., 2007.

[57] M. Fowler *et al*. (2011). Refactoring home page. [Online]. Available: http://www.refactoring.com/

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

[59] S. F. Smith and M. Thober, "Refactoring programs to secure information flows," in *Proc. the 2006 Workshop on Programming Languages and Analysis for Security*, Ontario, Canada, ACM, 2006.

[60] P. Li and S. Zdancewic, "Downgrading policies and relaxed noninterference," in *Proc. the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, NY, USA, pp. 158–170, ACM, 2005.

[61] M. Hafiz, "Security on Demand," PhD thesis, Graduate College of the University of Illinois at Urbana-Champaign, 2010.

[62] P. Joshi and R. K. Joshi, "Microscopic coupling metrics for refactoring," in *Proc. the Tenth European Conference on Software Maintenance and Reengineering*, R. K. Joshi, ed., p. 8, 2006.

[63] J. Kerievsky, *Refactoring to Patterns*, Addison Wesley, 2004.

[64] B. Alshammari, C. J. Fidge, and D. Corney, "A hierarchical security assessment model for object-oriented programs," in *Proc. the 11th International Conference on Quality Software (QSIC 2011)*, R. Hierons and M. Merayo, eds., Madrid, July 13-14, 2011.

[65] E. Tempero, "How fields are used in Java: An empirical study," in *Proc. Australian Software Engineering Conference*, pp. 91–100, 2009.

[66] E. Tempero, "An empirical study of unused design decisions in open source Java software," in *Proc. 15th Asia-Pacific Software Engineering Conference*, pp. 33–40, 2008.

[67] E. Tempero, J. Noble, and H. Melton, "How do Java programs use inheritance? an empirical study of inheritance in Java software," in *Proc. 22nd European Conference on Object-Oriented Programming*, J. Vitek, ed., vol. 5142, pp. 667–691, 2008.

Bandar M. Alshammari is an assistant professor at the Department of Information Technology at the University of Aljouf, Saudi Arabia. He earned his PhD from the Queensland University of Technology and his PhD title was "Quality metrics for assessing security-critical computer programs". He is currently the dean of admission and registration at the University of Aljouf.

Colin J. Fidge is a professor at the Science and Engineering Faculty at Queensland University of Technology, Brisbane, Australia. He earned his PhD from the Australian National University, Australia. He is currently the head of School of Electrical Engineering and Computer Science at Queensland University of Technology.

Diane Corney is currently the principal member of technical staff at Oracle, Brisbane, Australia. She worked as a senior research fellow at the Queensland University of Technology, Australia. She earned her PhD in 1997 from the Queensland University of Technology, Australia.