SOFTWARE SECURITY VULNERABILITY VS SOFTWARE COUPLING
A STUDY WITH EMPIRICAL EVIDENCE

A Thesis Presented to
The School of Computing and Software Engineering

by

Varadachari Sudan Ayanam

In Partial Fulfillment
of Requirements for the Degree
Masters of Science in Computer Science

Southern Polytechnic State University
December 2009
SOFTWARE SECURITY VULNERABILITY VS SOFTWARE COUPLING
A STUDY WITH EMPIRICAL EVIDENCE

Approved:

________________________________
Dr. Frank Tsui

________________________________
Dr. Venu Dasigi

________________________________
Dr. Han Reichgelt
In presenting this thesis as a partial fulfillment of the requirements for an advanced degree from Southern Polytechnic State University, I agree that the university library shall make it available for inspection and circulation in accordance with its regulations governing materials of this type. I agree that permission to copy from, or to publish, this thesis may be granted by the professor under whose direction it was written, or, in his absence, by the dean of the appropriate school when such copying or publication is solely for scholarly purposes and does not involve potential financial gain. It is understood that any copying from or publication of, this thesis which involves potential financial gain will not be allowed without written permission.

____________________________________
Varadachari Sudan Ayanam
Notice To Borrowers

Unpublished theses deposited in the Library of Southern Polytechnic State University must be used only in accordance with the stipulations prescribed by the author in the preceding statement.

The author of this thesis is:

Varadachari Sudan Ayanam
625 Floral Pl, Suwanee, GA 30024

The director of this thesis is:

Dr. Frank Tsui,
1100 South Marietta Pkwy, Marietta, GA 30060

Users of this thesis not regularly enrolled as students at Southern Polytechnic State University are required to attest acceptance of the preceding stipulations by signing below. Libraries borrowing this thesis for the use of their patrons are required to see that each user records here the information requested.

Name of user  Address  Date  Type of use (examination only or copying)
SOFTWARE SECURITY VULNERABILITY VS SOFTWARE COUPLING
A STUDY WITH EMPIRICAL EVIDENCE

An Abstract of
A Thesis Presented to
The School of Computing and Software Engineering

by

Varadachari Sudan Ayanam

In Partial Fulfillment
of Requirements for the Degree
Masters of Science in Computer Science

Southern Polytechnic State University
December 2009
ABSTRACT

Software Security has gained popularity and it is one of the buzzword in the field of Information Technology in the recent days. Software Coupling is another attribute that is of interest primarily to software developers, managers. Software Vulnerability refers to weakness or flaw in software that could be exploited to violate the system’s security.

In this Thesis the basic concepts and definitions of Software Security, Vulnerability, and Coupling will be discussed to lay out the theoretical foundation. It will then put forth the hypothesis that Software Coupling is a factor influencing Software Vulnerability. Later empirical data analysis will be done to prove the hypothesis that Software Coupling can influence Software Vulnerability. The empirical study is done using Mozilla’s source code as it is freely available to download and also it is one of the most popular internet suites. As a part of empirical analysis we put forth couple of security metrics that may be used to predict vulnerabilities in software.
SOFTWARE SECURITY VULNERABILITY VS SOFTWARE COUPLING
A STUDY WITH EMPIRICAL EVIDENCE

A Thesis Presented to
The Faculty of The School of Computing and Software Engineering

by

Varadachari Sudan Ayanam

In Partial Fulfillment
of Requirements for the Degree
Masters of Science in Computer Science

Advisor: Dr. Frank Tsui

Southern Polytechnic State University
December 2009
To Whom It May Concern

Dedicated to my father, my mother and my wife. Three people in my life who always believed in my potential and motivated me constantly.
All knowledge that the world has ever received comes from the mind; the infinite library of the universe is in our own mind.

Swami Vivekananda*
ACKNOWLEDGEMENTS

I would like to express my sincere thanks to my Thesis advisor Dr. Frank Tsui for his constructive criticism, inputs, help and support in writing this thesis. I would also like to thank my other committee members Dr. Ju An Wang and Dr. Sheryl Duggins for their support. I would take this opportunity to thank Dr. Edward Jung for his valuable comments and feedback. I would like to thank Dr. Venu Dasigi for his support in kick starting the thesis process and continuous help with departmental thesis procedures and format. I would like to express my sincere gratitude to Dr. Stephen Neuhaus, Saarland University, Saarbrücken, Germany, for sharing his valuable research data and materials which helped me a lot in writing this thesis. I would like to thank Mr. Baskar Parthiban for his support and motivation at my work. Last but not least, I would like to thank my wife Vaidhegi and daughter Tarkshika for their sacrifices.
# TABLE OF CONTENTS

**INTRODUCTION** .................................................................................................................. 2

**BASIC CONCEPTS AND DEFINITIONS** .............................................................................. 5

**SOFTWARE SECURITY** ........................................................................................................ 7
  - *Vulnerability and its classification* .................................................................................. 8
  - *Attackability* .................................................................................................................. 11
  - *Threats* ........................................................................................................................ 11
  - *Relation between Vulnerability, Threats, Attacks and Damages* .................................... 12

**COUPLING** .......................................................................................................................... 14

**SOFTWARE METRICS, NEED AND MEASUREMENT FRAMEWORK** ................................ 17
  - *Entities, Attributes and their Relationship* ................................................................. 21
  - *Units, scale types and their relationships* ..................................................................... 22
  - *Values* ......................................................................................................................... 23
  - *Measurement unit* ....................................................................................................... 24
  - *Indirect measures* ....................................................................................................... 24

**VULNERABILITY AND COUPLING** .................................................................................... 26

**SQL INJECTION** ................................................................................................................ 26
  - *SQL Injection Background* .......................................................................................... 26
  - *SQL Injection Techniques* .......................................................................................... 28
  - *SQL Injection Prevention* ............................................................................................ 30
  - *SQL Injection and Coupling* ....................................................................................... 31

**DENIAL OF SERVICE** ......................................................................................................... 32
  - *Denial of Service (Dos) Defined* ................................................................................ 32
  - *Denial of Service and Coupling* ................................................................................ 33

**BUFFER OVERFLOW ATTACKS** ....................................................................................... 34
  - *Types of Buffer Overflows* ........................................................................................ 35
  - *Program Stack* ............................................................................................................ 36
  - *Buffer Overflow Explained* ......................................................................................... 37
  - *Buffer overflow prevention* ......................................................................................... 38
  - *Buffer Overflow Attack and Coupling* ........................................................................ 40

**METHODOLOGY** .................................................................................................................. 42

**MEASUREMENT FRAMEWORK** .......................................................................................... 44
  - *Amount of Vulnerability in terms of Coupling Metric (AVC)* ....................................... 44
  - *Degree of Vulnerability in terms of Coupling Metric (DVC)* ......................................... 46
  - *Metric Weights* ........................................................................................................... 46
  - *Metric Value Interpretation* ......................................................................................... 48

**VULNERABILITY DATA COLLECTION** ................................................................................. 59

**COUPLING METRICS COLLECTION** .................................................................................. 62

**EMPIRICAL STUDY** .............................................................................................................. 64

**DATA ANALYSIS AND EVALUATION** .............................................................................. 64

**RESULTS** .............................................................................................................................. 71

**CONCLUSION** ...................................................................................................................... 73

**REFERENCES** ....................................................................................................................... 76
Chapter 1

INTRODUCTION

Every aspect of our lives is moving towards Information Technology. Software has become inevitable in our diurnal activities. Software quality is the key factor that is of interest to academician and industry professionals. Software quality attributes like reliability, maintainability, etc. have been studied well enough in the past. However, security has received less attention in the past but gained popularity recently due to pervasive computer networks, especially internet.

Today, security is one of the dominating news in IT. Disturbing titles, such as “Microsoft Security Advisory: Vulnerability in Microsoft Video Active X Control Could Allow Remote Code Execution” [9]; Twitter Crippled by Denial-of-Service Attack [26], are posted on the internet with alarming frequency. In spite of the many efforts in mitigating these security related problems [16 20], there is clearly room for more work.

Software attributes like coupling, cohesion, modularity and complexity are of interest only to the software developers but not to the external customers. These factors are well studied in few dimensions other than Software security. We would like to study software security and coupling to explore if these attributes are related.
In this thesis, we will first define and discuss the basic concepts related to terms such as software security, vulnerability, attackability, threats, damages, etc. These are related concepts and need to be clarified first. We will then define another major software characteristic, software coupling. It will become evident, at least at the conceptual level, that software coupling may be correlated to vulnerability, specifically security vulnerability. We will establish the rationale why we believe such a correlation may exist, or should exist. Furthermore, we will discuss why not all levels of coupling will contribute equally to security vulnerability and why certain type of coupling, such as content coupling, provides particularly fertile opportunity for security breaches.

Theoretical discussions and intuitions formulate a strong basis for what to look for in empirical data. However, the arguments will be greatly enhanced if there is real data to support the theoretical conjectures. We will use empirical data as evidential support in this thesis.

As part of the definitions and discussions of these fundamental terms and concepts, current literature will be surveyed. Many of the concepts will be drawn directly from the references. Some will be modified and strengthened. This will formulate Chapter 2 of the thesis. In Chapter 3 we will lay the ground work behind our belief of potentially strong correlation between certain types of coupling and certain security vulnerabilities. References to other people’s work in this area will also be cited. We then will generalize the belief to a formal hypothesis to be tested with empirical data. Chapter 4 will elaborate on the methodology and the sources of empirical data for the analysis. We then will analyze the data and show how software coupling characteristic can be an
influential factor to security vulnerability in Chapter 5. We will then conclude with some potential extensions and work for the future.
In this Chapter, we will discuss many of the fundamental concepts related to security. One of the goals is to formulate a more cohesive set of definitions that may be utilized in later chapters. We start with exploring what software security is.

Hogland and McGraw [4] define software security as the ability to defend attacker’s exploitation of software problems by building software to be secure throughout the development cycle. Software vulnerability is a weakness in a software system that allows an attacker to use the system for a malicious purpose. Attackability, according to Liu and Traore [5], is a measure of the extent that a software system or service could be the target of successful attacks. Thus attackability may be viewed as an indicator of vulnerability. The weakness or vulnerability can come from design flaws, implementation errors, configuration errors and other sources. Early detection and mitigation of vulnerabilities in software can help produce higher quality and secure software, prioritize testing efforts of these vulnerabilities and reduce the cost of later fixes.

Software security is also viewed as a software property. It belongs to a set of properties sometimes known as software quality. Software quality consists of combination of several attributes referred to as quality attributes. There are many perspectives in viewing these attributes. One way of viewing these properties is to
categorize them as: external and internal attributes.

- External attributes: these refer to software qualities that may be perceived as software performance related properties and may be of particular interest to all the stakeholders (e.g. users, customers, developers, etc). Reliability, maintainability, efficiency, compatibility and security are some of the external factors.

- Internal attributes: these correspond to hidden software qualities that are of interest mainly to software developers and professionals (e.g. developers, testers, maintainers, etc.). Readability, complexity, coupling and modularity are some of the internal factors.

External software quality attributes are influenced by internal attributes, and to improve software quality internal attributes needs to be affected. Internal attributes are easy to collect but hard to interpret while external attributes are easy to interpret but hard to collect. Prediction models allow the mapping and projection of hard-to-interpret internal measurement data into easily interpretable external measurement data. We believe that one internal attribute that affects software security qualities (via vulnerability) is coupling. Coupling poses a direct threat to software vulnerability.

The purpose of our study is to find out if Software Coupling, an internal attribute, and Software Vulnerability, an external attribute, are correlated and to show empirical
evidence between them. There have been few empirical evidences shown between Coupling and individual vulnerabilities like DOS attackability [5], etc. But none of them have claimed the relation between Coupling and Vulnerability in general.

**Software Security**

As mentioned earlier, one of the definitions for software security is the ability to defend attacker’s exploitation of software problems by building software to be secure throughout the development cycle [4]. We will modify this definition and provide our own, a more formal, version later in this Chapter.

Another definition is from National Institute of Standards and Technology (NIST). Guttman and Roback stated in their NIST Report that security is the protection given to the system such that integrity, availability and confidentiality of the system resources are preserved [25]. They further elaborate that:

- Confidentiality is preventing unauthorized entity from access of information/system,

- Integrity is the ensuring of only authorized entity can manipulate the information/system in a predefined and authorized way and

- Availability is the capability of access of information/system by authorized entity on all appropriate occasions.

Note that this definition moves security from the realm of non-functional properties to the functional capabilities of “prevention.”
Software Security is one of the important external quality attributes that is of concern to Software developers and end users these days. Vulnerability evaluation plays a central role for security posture and risk management.

**Vulnerability and its classification**

In this section we will explore the term, vulnerability, in some depth. Ivan Krsul defined software vulnerability as an instance of a “fault” in the specification, development, or configuration of software such that its execution can violate an [implicit or explicit] security policy [7]. Stated a bit differently, vulnerability refers to flaws or weakness in a system’s design, implementation, or operation and management that could be exploited to violate the system’s security policy [6]. Any flaw or weakness in an information system could be exploited to gain unauthorized access, to inflict damage or to compromise the information system [6].

Software vulnerability is a weakness in a software system that allows an attacker to use the system for a malicious purpose. Buffer overflow, SQL injection, cross-site scripting are some of the representative vulnerabilities [8].

Vulnerability is an unfortunate characteristic, a flaw that allows a threat to potentially occur [2]. Associated with the vulnerability concept is the notion of a “threat.” Amoroso defined a threat as any potential occurrence, malicious or otherwise, that can have an undesirable effect on the assets and resources associated with a computer system.
Vulnerability is a bug, flaw, weakness, or exposure of an application, system, device, or service that could lead to a failure of confidentiality, integrity, or availability. These potential failures may be viewed as threats.

Vulnerability, itself, may be broadly classified into three main categories:

- Software flaws,
- Security configuration issues and
- Software feature misuse

Software flaw vulnerability is caused by an unintended error in the design or coding of software. An example is an input validation error, such as user-provided input not being properly evaluated for malicious character strings and overly long values associated with known attacks.

A security configuration setting is an element of software’s security that can be altered through the software itself. Examples of settings are an operating system offering access control lists that set the privileges that users have for files, and an application offering a setting to enable or disable the encryption of sensitive data stored by the application. Security configuration issue vulnerability involves the use of security configuration settings that negatively affect the security of the software.

A software feature is a functional capability provided by software. Software feature misuse vulnerability is a vulnerability in which the feature also provides an
avenue to compromise the security of a system. These vulnerabilities are caused by the software designer making trust assumptions that permit the software to provide beneficial features, while also introducing the possibility of someone violating the trust assumptions to compromise security. For example, email client software may contain a feature that renders HTML content in email messages. An attacker could craft a fraudulent email message that contains hyperlinks that, when rendered in HTML, appear to the recipient to be benign, but actually take the recipient to a malicious web site when they are clicked on. One of the trust assumptions in the design of the HTML content rendering feature was that users would not receive malicious hyperlinks and click on them.

There are many types of vulnerabilities. Buffer overflows, SQL injection, Denial of Service attackability, URL Jumping are some of the representative vulnerabilities. These types of vulnerabilities may be classified into the above mentioned 3 main categories. For example, buffer overflow may be a software flaw while SQL injection may be a feature misuse.

We can see that the property vulnerability as defined here has an inverse relationship to security. That is, security embraces the notion of “defense” or “protection.” Vulnerability, on the other hand addresses the “bugs” or “flaws” that lower or lessen such “protection.”
**Attackability**

If software has some vulnerability, then it is potentially open to attacks. An attack on a computer system is some action taken by a malicious intruder that involves exploitation of certain vulnerabilities in order to cause an existing threat to occur [2]. Attacks are often heuristic, involving some knowledge about vulnerabilities on the part of the attacker and it usually results in damages to the system under attack.

**Threats**

Earlier we stated that, according to Amoroso, a threat is any potential occurrence, malicious or otherwise, that can have an undesirable effect on the assets and resources associated with a computer system. Note that a threat can be associated with one or more vulnerabilities, and multiple threats may be associated with a single vulnerability. Also, multiple attack methods can be associated with a given vulnerability.

Amoroso [2] further defines the following major types of threats:

- Disclosure Threat,
- Integrity threat and
- Denial of service threat

Disclosure threat involves the dissemination of information to an individual for whom that information should not be seen. In terms of computer security, the disclosure threat occurs whenever some secret on a computer system or in transit between systems is
shown to someone who should not know the secret. The term “leak” is often associated with Disclosure threat. Confidentiality is other term associated with Disclosure threat.

The integrity threat involves any unauthorized change to information on a computer system or in transit between computer systems. It is important to point out that a legitimate change may be made by an unauthorized entity and that an illegitimate change may be made by an authorized person. Both situations pose an integrity threat. When intruders maliciously alter information, we say that the integrity of the information has been tampered with.

Denial of service threat arises whenever access to some computer system resource is intentionally blocked as a result of malicious action taken by another user. If one user requires access to a service and another user does something malicious to prevent such access, then denial of service has occurred. Availability is a commonly used term that is associated with Denial of Service threat.

**Relation between Vulnerability, Threats, Attacks and Damages**

The following real world example by Amoroso [2] will help in understanding threats, vulnerability and attacks better. Consider that a threat associate with a house is that a burglar might steal furniture, money, etc. A possible vulnerability might be an open window and an attack method might be to climb through the window.
It is clearly understandable from the above definitions that, vulnerable software exposes a threat that can be attacked by malicious users. Attacks result in damage to the system/software that is vulnerable. Attack resulting in damage can be understood with relative ease rather than attacks in terms of threats. In this thesis we will focus on damage than threat as the term damage is easier to interpret and understand. Vulnerable software could lead to a potential compromise of software security even though the vulnerability has not been attacked yet. Finding and fixing the vulnerabilities is key to make a software secure and to avoid damages to software/system.

**Formal Definitions of Software Vulnerability Security and their Relation**

We will adopt the concepts of threat, attacks, vulnerability, and security from the above literature survey and provide a more crisp set of definitions here. First we define the concept of software vulnerability as follows.

**Definition 1:**

Software vulnerability is a software defect caused by a design, implementation, or configuration error which will allow external intervention to the software to behave in unintended ways. These external interventions are usually designed with malicious intentions.

Next we define the concept of software security in terms of the above defined vulnerability as follows.
Definition 2:
Software Security is a software attribute that describes the extent of software vulnerability.

If the system is vulnerable then it is open to attack resulting in damages that affect integrity, availability and confidentiality. As the system becomes more vulnerable its security decreases and vice versa. Thus security is inversely proportional to vulnerability. Thus, we can now view software security as an attribute which may be measured by the amount or extent of vulnerability that exists in that software. In Chapter 4, we will further explore this relationship and the resulting measurement.

**Coupling**

Software Coupling is an internal software attribute that describes the degree of dependence between software components. Loosely coupled components have less dependence between them, while highly coupled components have great deal of dependence between them. Uncoupled components have no interconnections at all. In reality, it is unlikely that a system can be built of completely uncoupled components [3]. Thus the goal of software engineers is not necessarily complete independence, but rather keeping the degree of coupling as low as possible.

Traditionally, software coupling is broken into five categories, listed here in order from best to worst type of coupling.
- data coupling
- stamp coupling
- control coupling
- common coupling
- content coupling

Data coupling is the type of coupling where data is the only factor of dependence. If only the needed data is passed among the components, then the components are connected by data coupling. Data coupling is simpler and leaves less room for error. Data coupling is the most desirable form of coupling as it is easy to trace and make changes.

Stamp Coupling is the type of coupling when an entire data structure is shared between components, then the components are connected by stamp coupling. Note that in stamp coupling, there is a possibility of passing more than the absolutely needed data. Stamp coupling is next sought after Data Coupling.

Control Coupling is the type of coupling when one component passes parameters to control the activity of another component, then the components are said to exhibit control coupling. Control coupling is preferred after Data and Stamp coupling.
Common coupling is the type of coupling that occurs when two modules share the same global data (e.g. a global variable or a complete DB table). This is preferred next to Data, Stamp and Control Coupling.

Content Coupling is the type of coupling when one component modifies the internal data item in another component, then the components are said to be content coupled. Content coupling is the least preferred type of coupling.

Alghamdi [21] viewed coupling as binary relations from R5 to R0 and is shown below in order of least to most preferred type of coupling. The actual definitions of the relationship are not any different from the above definitions of coupling. The implication here is that there is an implicit ordering Ri < Ri+1, where the symbol, <, semantically means “less desirable.” Thus Ri+1 is less desirable than Ri.

Content coupling relation R5: (x, y) ∈ R5 if x refers to the internals of y, i.e., it branches into, changes data, or alters a statement in y.

Common coupling relation R4: (x, y) ∈ R4 if x and y refer to the same global variable.

Control coupling relation R3: (x, y) ∈ R3 if x passes a parameter to y that controls its behavior.

Stamp coupling relation R2: (x, y) ∈ R2 if x passes a variable of a record type as a parameter to y, and y uses only a subset of that record.
Data coupling relation R1: \((x, y) \in R1\) if \(x\) and \(y\) communicate by parameters, each one being either a single data item or a homogeneous set of data items that does not incorporate any control element.

No coupling relation R0: \((x, y) \in R0\) if \(x\) and \(y\) have no communication, i.e., are totally independent.

Coupling is one of the important internal quality attributes that affects overall software quality. Coupling is related to sharing software entities and components. When information is passed among components there is high potential that it is also “exposed.” That is, there is a potential that the information may be captured and altered by others. Coupling may be considered a “flaw” in the design or as software vulnerability according to our Definition 1. Thus, intuitively coupling can be a factor that can affect software security. However without substantial evidence this cannot be concluded.

**Software Metrics, Need and Measurement Framework**

In showing relationship between two attributes, namely security and coupling, we will be using measurements of these attributes. Thus it is important that the metrics we employ are based on some theoretical foundation. We strongly feel that it would be appropriate to touch upon measurement and metrics theory at this point to better understand the vulnerability based security metric that we will propose in chapter 4.
For an Engineering discipline to sustain its existence it should mature over time. As it matures it needs to be measured to understand further and to be controlled. Without proper metrics in place measurement is not possible. By proper metrics we mean a valid metrics. Measures must represent accurately those attributes they purport to quantify [27]. So validation is critical to the success of software measurement [27].

Software engineering is not an exception either. Valid software metrics that enables it to understand, monitor and control development processes and products are needed. Software measurement must be consistent with measurement in other disciplines. Indeed, any validation framework that contradicts measurement principles that apply to other disciplines would itself be invalid.

Schneidewind [28] recommends an empirical validation process by which a software metric is validated by showing that it is associated with some other measure of interest. Fenton and Kitchenham [29] discussed two different views of validation: one based on identifying the usefulness of a measure for predictive purposes, the other on identifying the extent to which a measure characterizes a stated attribute. In this thesis we use empirical validation method to characterize the security attribute to support our claim.
Barbara Kitchenham, Shari Lawrence Pfleeger, Fenton [27], proposed a measurement framework that helps to validate a measure, assess the validation work of others and to apply a measure in a given situation. This framework is shown in Figure 1.1 where real world entity attributes are measured formally using measurement principles and expressed in units of scale types i.e. nominal, ordinal, interval and ratio. Figure 1.2 talks about the legends used in figure 1.1.

In Figure 1.3, we have instantiated the general terms in figure 1.1 with the specific for capturing the measurement of the specific attribute, security. For instance, the box labeled “Entity” is now changed to “Software”, the box labeled “Attribute” is changed to “Security”, “Value is denoted with real values “1 to n”, “Unit” captured as “extent of vulnerability”, “Measurement Instrument” shown as “counting the number of weighted types of vulnerability “ and the “scale type” used is “ordinal”. Our measurement model captures the gist of this framework by trying to map real world attributes, Coupling and Vulnerability, to formal world in coming up with the metrics. The structural model in figure 1.3 will be the framework for introducing the specific Vulnerability and Security metrics in Chapter 4.
Figure 1.1

Empirical (Real) World

Entity
  possesses
  Attribute (Dimension)
    measures
    quantifies

Formal (Mathematical) World

Value
  determined by
  expressed_in
  uses
  belongs_to
  Scale Type

Figure 1.2

Key to Figure

<table>
<thead>
<tr>
<th>entity</th>
<th>Entity name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ea1</td>
</tr>
</tbody>
</table>

1 to many relationship

1 to many optional relationship

Relationship with arrow
Showing direction

relationship_name

Ea1
r1
Ea1

Ea1
r2
Ea1
Entities, Attributes and their Relationship

Entities

Entities are the objects we observe in the real world. Software entities may be products, processes, or resources of different types. In our case it is the software quality

Attributes

Attributes are the properties that an entity possesses. Ex: security, reliability, coupling, etc. For a given attribute, there is a relationship of interest in the empirical
world that we want to capture formally in the mathematical world. Properties are often multi-dimensional. Multi-dimensional attributes can be vectors, e.g., velocity which involves speed and direction or scalars, e.g., speed which is measured in terms of distance per time period. These distinctions are often ignored in software measurement and many measurement problems could be overcome if they were properly understood.

**The Relationship between Entities and Attributes**

An entity possesses many attributes, while an attribute can qualify many different entities. Consider a program as a software entity which can exhibit attributes such as length, structure, and correctness. On the other hand an attribute like height applies to human beings, mountains, and houses.

**Units, scale types and their relationships**

**Units**

A measure maps an empirical attribute to the formal, mathematical world. A measurement unit determines how we measure an attribute, and an attribute may be measured in one or more units. For example, you may use different units to measure temperature (e.g., Fahrenheit, Celsius, or Kelvin). The same unit may be used to measure more than one attribute. For example, fault rate may be used to measure the program correctness or test case effectiveness.
**Scale Types**

The most common scale types are: nominal, ordinal, interval and ratio. A unit’s scale type determines the admissible transformations we can apply when we use a particular unit [27].

In the context of nominal and ordinal scale measures where our measures are mappings to arbitrary labels, we suggest a “unit” is needed to ensure that such measures are used consistently.

For example, if we are defining fault categories as major, minor and negligible, we need to define these terms in more detail if different data collectors are going to use the terms consistently. In this case our “units” would be the description of major (e.g., a fault that result in a software failure), minor (e.g., a fault that results in misleading or unhelpful outputs to the user), and negligible (e.g., a code structure that conflicts with standard coding practices).

**The Relationship between Units and Scale Types**

The scale type is inherent in the unit not the attribute. For example, Fahrenheit and Celsius are interval scale units of temperature, whereas Kelvin is a ratio scale unit of temperature. Thus, although the different units lead to different scale types, they do not affect the attribute.

**Values**

When we measure an attribute, we do so by applying a specific measurement unit to a particular entity and attribute to obtain a value. A measured value cannot be
interpreted unless we know to what entity it applies, what attribute it measures and in what unit. Just as a price is always associated with a specific item, and a unit of currency (e.g., dollars, guilders, pounds), so must a software attribute have both an entity and a unit of measure; one or two without the third are meaningless. We expect valid measures to be defined over a set of permissible values; for example, length in lines of code is defined on the non-negative integers.

**Measurement unit**

There may be many different measurement instruments available for a particular unit. For example, we can measure height by using either a tape measure or variations in air pressure. Measurement instruments usually detect a single (unit) value of an attribute in a particular unit of measurement and accumulate units into a value for a particular entity. However, measurement instruments are also used to classify entities. For example, we might use a genetic test as a measurement instrument to determine the sex of an athlete.

**Indirect measures**

We often obtain measures from equations involving other measures. Such measures are called indirect measures. The equation defining an indirect measure acts as a form of measurement instrument. The attribute(s) we use in an equation may relate to an entity or entities different from the entity whose attribute we want to measure indirectly. For example, although we may use an equation to predict effort from size, size is a product attribute whereas effort is a process or a resource attribute.
In this thesis, we measure software security indirectly in terms of software vulnerability.

The attributes security and vulnerability are associated using inverse relationship. The application of the measurement frameworks and the association between them and the metric we are going to propose will be discussed in detail in chapter 4.

The following are the key points concluded by Barbara Kitchenham, Shari Lawrence Pfleeger, and Fenton [27]

- Use measures that you understand in the context of your own goals and situation.
- If you are concerned about a multi-dimensional attribute such as complexity or quality, use different measures for different aspects of the attribute.

In this thesis we have devised metrics for direct measurement of the attribute, software coupling, which in turn will be used as an indirect measure of software vulnerability. Vulnerability is measured via types and amount of coupling in software. As discussed earlier we use software vulnerability as an indirect measure for software security as software vulnerability and software security are inversely related. The metrics will be used primarily in empirical verification of relationship between coupling and security and it may be possibly used for prediction.
Chapter 3

VULNERABILITY AND COUPLING

Vulnerability is a defect that exposes parts of the software information or behavior in such a manner that could open up possibility of attacks. In this chapter we will discuss about some individual types of attacks and their possible correlation to coupling.

SQL Injection

SQL Injection Background

Web applications employing database-driven content are ubiquitous. Companies whose business model focuses on the Internet, such as Yahoo and Amazon, are obvious examples; but nearly every major company has a web presence, and this presence often uses a relational database. Various technologies like Sun J2EEE, Microsoft .NET, etc are available that can provide frameworks for web applications. These technologies typically employ three tiers: a) the presentation tier, b) middle tier, and c) data tier. The presentation tier uses a web browser to capture user input and present information output. This is typically written in HTML, but there can be some other custom thin client. The middle tier, also known as the business tier, encapsulates the business logic that drives the application. This software layer is responsible for constructing information requests to
the database layer. The database tier is typically a relational database and provides storage services.

The web browsers i.e. clients, accept inputs from users in the form of username and password typed in to the webpage i.e. presentation tier. The web page posts this information as name: value string pairs (input field: value) to the middle tier. The middle tier uses this input to create a query, or request, to the database layer. This is almost always SQL (Structured Query Language), such as SELECT * FROM users WHERE username='temp' AND password='secret'. The data layer processes the request and returns a record set back to the middle tier. The middle tier manipulates the data, creates a session, and then passes a subset to the presentation tier. The presentation tier renders the information in HTML for the browser. Notice that the middle layer generates an SQL request based on input supplied by the user. Because of the popularity of these web applications, techniques to exploit their security vulnerabilities are potentially quite dangerous. One such technique is called SQL injection.

SQL injection is a type of security attack whereby a malicious user’s input is parsed as part of the SQL statement sent to the underlying database [10]. The goal of the attack is to query the database in a manner that was not the intent of the application programmer. SQL Injection Attack (SQLIA) occurs when an attacker changes the developer’s intended structure of an SQL command by inserting new SQL keywords or operators [14].
A *SQL injection attack* is performed when a user exploits a lack of input validation to force unintended system behavior by altering the logical structure of a SQL statement with special characters [15].

SQL injection attacks (SQLIAs) are one of the major security threats for Web applications [11]. Two well-known companies whose public web sites were vulnerable to such attacks are Guess and Petco. Both vulnerabilities were discovered by a curious 20 year old programmer in 2003. As a result, Petco exposed 500,000 credit cards, and required a settlement with the Federal Trade Commission [12, 13].

**SQL Injection Techniques**

Several SQL injection techniques will be explored in this section. These may all be considered SQL injection attacks.

**Tautologies**

One method to gain unauthorized access to data is to insert a tautology into the query. In SQL, if the WHERE clause of a SELECT or UPDATE statement is disjointed with a tautology, then every row in the database table is included in the result set.

To illustrate, consider an online banking application. Assume that a user has logged into the web site properly. To update their account information, the user navigates to the proper page using a (generated) query string where the user id appears, for example: details.asp?id=22. The value of the id name-value pair (i.e., the string 22) is
then used by the middle tier to generate the SQL statement SELECT * FROM users
WHERE userid=22, and the appropriate information is returned back to the user.

However, an attacker could easily manipulate this interaction by editing the query string
to contain a tautology, such as details.asp?id=22 OR 1=1. Again, the value of id is used
(i.e., the string 22 OR 1=1) in constructing the SQL statement, which becomes SELECT
* FROM users WHERE userid=22 OR 1=1. This query returns all rows of the users table.

**UNION Queries**

Another SQL injection technique involves the UNION keyword. SQL allows two
queries to be joined and returned as one result set. For example, SELECT col1, col2, col3
FROM table1 UNION SELECT col4, col5, col6 FROM table2 will return one result set
consisting of the results of both queries. Let us return to our previous example, SELECT
* FROM users WHERE userid=22. If the attacker knew the number and types of the
columns in the first query, an additional query such as SELECT body, results FROM
reports can be appended. For some applications, surmising this information is not
difficult. If the programmer is not consuming all exceptions, incorrect SQL queries will
generate error messages that expose the needed information [10]. For example, if the two
queries in the UNION clauses have a disparate number of columns, an error such as all
queries in an SQL statement containing a UNION operator must have an equal number of
expressions in their target lists will be returned. The attacker merely changes the query to
SELECT * FROM users WHERE userid=22 UNION SELECT body, results, 1 FROM
reports to try and match the number of columns. The attacker can continue to add dummy
columns until an error such as syntax error converting the varchar value 'txfrs' to a
column of data type int occurs. This signals that the column count is correct, but at least one column type is not. The attacker can then vary the types accordingly.

**Using Comments**

SQL supports comments in queries. Most SQL implementations, such as T-SQL and PL/SQL use two dashes, --, to indicate the start of a comment (although occasionally # is used). By injecting comment symbols, attackers can truncate SQL queries with little effort. For example, SELECT * FROM users WHERE username='greg' AND password='secret' can be altered to SELECT * FROM users WHERE username='admin' -- AND password='". By merely supplying admin -- as the username, the query is truncated, eliminating the password clause of the WHERE condition. Also, because the attacker can truncate the query, the tautology attacks presented earlier can be used without the supplied value being the last part of the query. Thus attackers can create queries such as SELECT * FROM users WHERE user-name='anything' OR 1=1 -- AND password='irrelevant'. This is guaranteed to log the attacker in as the first record in the users table, often an administrator.

**SQL Injection Prevention**

Litwin [16] describes five measures that you can take to prevent SQL injection attacks. The author suggests that you implement as many of these measures as possible to have multiple layers of security in your application. That way if one of the measures is circumvented because of some vulnerability, you are still protected. First, you should never trust user input. You should never use input from a database query that has not
been validated. According to the author, the best approach to validate user input is to “identify the allowable characters and allow only those characters.” Second, you should never use dynamic SQL. SQL injection attacks are dependent on dynamic SQL queries. The author suggests using stored procedures or SQL queries that accept parameters. Third, you should never connect to a database using an admin-level account. Fourth, don’t store passwords in plain text. The author suggests that you encrypt or hash passwords, encrypt connection strings and other sensitive data. Fifth and finally, error messages that the users see should display minimal information.

**SQL Injection and Coupling**

Any batch of SQL generated dynamically by an application layer and sent to SQL server for execution is called ad hoc SQL [17]. Many web applications use ad hoc SQL these days and there has been an industry wide argument on the usage of ad hoc SQL vs stored procedures. Both the approaches have their own benefits, however both Litwin [16] and Adam, etal [17] claim that using ad hoc SQL could lead to security holes in the software.

As the various SQL injection techniques have shown, ad hoc SQL presents various security challenges including opening possible attack vectors and making data access security much more difficult to enforce [17]. By using ad hoc SQL your application may be more vulnerable to being hacked [17]. The end result is that a greater degree of testing will be required in order to ensure that security holes are properly
patched and both authorized and non-authorized are unable to access data they are not supposed to see. These ad hoc SQL are same as dynamic coupling.

As shown in Figure 2.1 below, coupling between application and database layers is the key factor involved in the use of ad hoc SQL.

![Diagram](image)

**Figure 3.1: Control Coupling with SQL String Passing**

As pointed out earlier that ad hoc SQL, thus dynamic coupling, specifically control coupling, opens up security holes. The resulting vulnerability provides opportunity for attacks. By using stored procedures with correctly defined interfaces and full encapsulation of information, coupling between the application and the database will be greatly reduced [17]. This encapsulation can possibly reduce security risks.

**Denial of Service**

**Denial of Service (Dos) Defined**

DoS attack is designed to render a computer or network incapable of providing normal services [18]. The most common DoS attacks will target the computer's network bandwidth or connectivity. Bandwidth attacks flood the network with such a high volume
of traffic that all available network resources are consumed and legitimate user requests can not get through. Connectivity attacks flood a computer with such a high volume of connection requests, that all available operating system resources are consumed and the computer can no longer process legitimate user requests.

If computer hardware, software, and data are not kept available, productivity can be degraded, even if nothing has been damaged. Denial-of-service can be conceived to include both intentional and unintentional assaults on a system's availability. The most comprehensive perspective would be that regardless of the cause, if a service is supposed to be available and it is not, then service has been denied. An attack, however, is an intentional act. A denial-of-service attack, therefore, is considered to take place only when access to a computer or network resource is intentionally blocked or degraded as a result of malicious action taken by another user [19]. These attacks don't necessarily damage data directly, or permanently (although they could), but they intentionally compromise the availability of the resources.

DoS attacks are also one of the major security threats for Web applications [22].

**Denial of Service and Coupling**

There had been a number of attacks on apache web servers like Apache MIME flooding [18] or Apache Sioux Attack [18]. With specially formatted HTTP requests it was possible to make the web server use up huge amounts of memory. As soon as the system started swapping out server processes or used up the whole system memory, the
server was effectively dead. These attacks target web server by exploiting implementation bugs. Sending out specially formatted HTTP request that can control the actions of web server is a prime example of Coupling (e.g. control/content coupling). In a way, this is similar to altering the SQL statement in SQL injection. Here the HTTP message is altered.

Frank, et al. [18] also claim that unauthorized and invalidated foreign access to database servers should be blocked to avoid DoS attacks. This is another evidence of coupling between database servers and application leading to DoS attack.

Microsoft's Internet Information Server versions 4.0 and 5.0 suffered from a so called server URL parsing bug. The decoding of escape sequences in URL strings was implemented very inefficiently. Submitting long strings with large amounts of escape characters effectively stopped the web server from working for a significant time (DoS attack). A fix is available from Microsoft [20]. This is yet another security hole opened up by coupling (content) between web server and client application.

**Buffer Overflow Attacks**

Buffer overflow attacks are one of the most common attacks on computers. The year 2003 saw about 75% of all CERT advisories come from buffer overflow vulnerabilities [30]. Worms, such as the Code Red [31] and Slammer [32] spread incredibly quickly and had the potential to be completely devastating. Writing code that
attempts to avoid such vulnerabilities is not always sufficient, especially due to the abundance of vulnerable existing and legacy codes.

According to Microsoft [33], a buffer overflow attack is an attack in which a malicious user exploits an unchecked buffer in a program and overwrites the program code with their-own data. If the program code is overwritten with new executable code, the effect is to change the program's operation as dictated by the attacker. If overwritten with other data, the likely effect is to cause the program to crash.

A buffer is a contiguous allocated chunk of memory, such as an array or a pointer in C.

Buffer overflows have assumed several different names over the years. These include buffer overrun, stack overrun, and stack overflow. In practice, all these terms share the same definition and can be used synonymously and interchangeably. Additionally, the stack buffer overflow exploit is often referred to as “stack smashing” in modern day parlance.

**Types of Buffer Overflows**

The “Stack” and the “Heap” overflows are the two primary types of buffer overrun situations [33]. The stack overflow has two basic variations. One type involves overwriting (and thus changing) security sensitive variables or control flags stored in memory adjacent to the unchecked buffer. The most common type of stack overflow
involves the overwriting of function pointers that can be used to change program flow or gain elevated privileges within the operating system environment. The more complex heap overrun involves dynamic memory allocations, or memory allocated at run time by an application. Stack buffer overflow is more prevalent and much popular and we will discuss about it in the following sections.

In simpler terms, a buffer overflow is very much like pouring ten ounces of water in a glass designed to hold eight ounces. Obviously, when this happens, the water overflows the rim of the glass, spilling out somewhere and creating a mess. Here, the glass represents the buffer and the water represents application or user data.

**Program Stack**

To better understand the stack buffer overflow, it would be appropriate to discuss about program stack structure.

A stack is a contiguous block of memory containing data. A stack pointer (SP) points to the top of the stack. Whenever a function call is made, the function parameters are pushed onto the stack from right to left. Then the return address (address to be executed after the function returns) followed by a frame pointer (FP), is pushed on the stack. A frame pointer is used to reference the local variables and the function parameters, because they are at a constant distance from the FP. Local automatic variables are pushed after the FP. In most implementations, stacks grow from higher memory addresses to the lower ones.
Let’s look a simple C code snippet to understand the stack region.

```c
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}

void main() {
    function(1,2,3);
}
```

For the above code snippet the stack region will look like this

```
bottom of                                          top of
memory                                            memory
        buffer2      buffer1      sfp      ret      a      b      c
<--------                  [    ]    [    ]    [    ]    [    ]    [    ]    [    ]
```

**Buffer Overflow Explained**

In C and C++, there are no automatic bounds checking on the buffer, which means a user can write past a buffer [34].

Consider another C example that can overrun the buffer:
void function (char *str) {
    char buffer[16];
    strcpy (buffer, str);
}

int main () {
    char *str = "I am greater than 16 bytes"; // length of str = 27 bytes
    function (str);
}

This program is guaranteed to cause unexpected behavior, because a string (str) of 27 bytes has been copied to a location (buffer) that has been allocated for only 16 bytes. The extra bytes run past the buffer and overwrite the space allocated for the FP, return address and so on. This, in turn, corrupts the process stack. The function used to copy the string is strcpy, which completes no checking of bounds. Using strncpy would have prevented this corruption of the stack. However, this classic example shows that a buffer overflow can overwrite a function's return address, which in turn can alter the program's execution path. Recall that a function's return address is the address of the next instruction in memory, which is executed immediately after the function returns.

**Buffer overflow prevention**

The following are some of the methods that are proposed in various literatures

[30, 33, 34]
Write Secure Code

Buffer overflows are the result of stuffing more code into a buffer than it is meant to hold. C library functions such as strcpy(), strcat(), sprintf() and vsprintf() operate on null terminated strings and perform no bounds checking. gets() is another function that reads user input (into a buffer) from stdin until a terminating newline or EOF is found. The scanf() family of functions also may result in buffer overflows. Hence, the best way to deal with buffer overflow problems is to not allow them to occur in the first place. Developers should be educated about how to minimize the use of these vulnerable functions and availability of alternate functions like strncpy, etc.

Stack Execute Invalidation

Because malicious code (for example, assembly instructions to spawn a root shell) is an input argument to the program, it resides in the stack and not in the code segment. Therefore, the simplest solution is to invalidate the stack to execute any instructions. Any code that attempts to execute any other code residing in the stack will cause a segmentation violation.

Compiler Tools

Over the years, compilers have become more and more aggressive in optimizations and the checks they perform. Various compiler tools already offer warnings on the use of unsafe constructs such as gets(), strcpy() and the like. Choosing proper
compiler tools and options can throw warnings to the user on potential buffer overflow attacks.

**Static code analysis**

Use of static code analysis tool in the development is encouraged to find out possible buffer overflow attacks. There are various tools like ITS4, RATS, LCLint and PurifyPlus available that can find out possible buffer overflow vulnerabilities present in the code.

**Buffer Overflow Attack and Coupling**

Code Red I version 1 was released on July 12th, 2001 [35][36], approximately a month after the discovery of this vulnerability by eEye [37]. After its release it was then analyzed by eEye [38]. The worm used an HTTP GET request to overflow the buffer. In the request, the query string was used to overflow the buffer and inject the code. Once infected, the Code Red I worm generated 100 threads with which to randomly scan for more vulnerable IIS servers. The various examples of buffer overflow attacks result from the existence of content coupling between clients and server where one program (client) influences the content of another program (server). This is dangerous even when there is no malicious attempt.

Most of the web servers are written in C, C++ to for performance reasons. Any invalidated input from the client to the web browser can be exploited for buffer overflow
attacks on these web servers. Once again this is possible because of coupling between web servers and clients.

As systems complexity and size increases, coupling is inevitable. With 3-tiered web applications, coupling is almost built into the design style of model-view-control (MVC). However, content coupling, common coupling, and control coupling are especially dangerous. These types of coupling are the sources of security vulnerabilities that may lead to SQL injection, DoS attacks or Buffer overflow attacks.
Chapter 4

METHODOLOGY

The objective of this thesis is to explore if software coupling could be a significant factor influencing software security vulnerability and to show empirical data evidence for the relation between coupling and vulnerability by proposing new metrics. As discussed earlier in Chapter 2, Security can be expressed in terms of vulnerability as they are inversely proportional.

Even though intuitively coupling seems to affect software security it is not enough to claim so without any empirical evidence. Several authors [5, 23] have shown relationship between Software Coupling and some individual types of security vulnerabilities like Denial-of-Service attackability, URL Jumping. But to my knowledge no one has ever claimed coupling to be a factor of vulnerability in general. This thesis will try to claim this in general by showing empirical data evidence with the use of software coupling metrics. We state our main hypothesis as follows

**Hypothesis H1:**

**Software Coupling is a factor influencing Software Security Vulnerability**

If H1 is true then Software Coupling metrics, which is relatively easier to collect, can be used to identify vulnerable components. The same metrics can then be used to predict software vulnerabilities.
To show that H1 is true, a new metric is proposed in this Chapter using Software Coupling metrics together with Vulnerability bugs from Mozilla vulnerability database. Coupling metrics are collected using Understand C++ [24] statistical analysis tool and Mozilla vulnerability bug report is obtained from [1].

Mozilla firefox is used because it is an open source project for which its source, vulnerability information are available freely. Also Mozilla is one of most popular and widely used internet application suites. Moreover, Stephen Neuhaus [1], etal have also used Mozilla database to come up with their findings and their findings will be used in our security metrics in this thesis.

Stephen Neuhaus, etal in their article titled “Predicting Vulnerable Software Components” [1] introduced a new tool called Vulture, a new approach and tool to predict vulnerable components in large software systems. Vulture relates a software project’s version archive to its vulnerability database to find those components that had vulnerabilities in the past. It then analyzes the import structure of software components and uses a support vector to learn and predict the vulnerable components. Using Vulture the authors have identified the top 10 most vulnerable components of Mozilla and this data will be used as input to this thesis to prove our hypothesis.
**Measurement Framework**

Software Security is an attribute that can be described in terms of Vulnerability. As vulnerability increases software security decreases. Also we have shown theoretical association between Coupling and Vulnerability, and intuitively coupling and vulnerability seem to be related. The real challenge lies in coming up with a common metric that can capture the relation between coupling and Vulnerability (Security in turn). Coupling metrics are easy to collect and hard to interpret, and the hardest part lies in interpreting coupling in terms of vulnerability. On the other hand vulnerability is easy to interpret but hard to collect. In our case the vulnerability metrics are derived using data obtained from Neuhaus, etal [1] study of Mozilla database. Even though our Chapter 1 and Chapter 2 literature survey captures software vulnerability in terms of coupling, without metric and data evidence the literature survey becomes incomplete.

**Amount of Vulnerability in terms of Coupling Metric (AVC)**

As pointed out earlier, vulnerability may be measured in terms of coupling via types and amount of coupling. There are multiple types of coupling, and we capture their contributions to vulnerability by taking all of them into account through a weighted average metric, called Amount of Vulnerability via Coupling, AVC. AVC metric is used for this purpose.
AVC metric is defined using the traditional five types of coupling together with the weights based on the individual coupling preferences.

\[ AVC = W_1 \times C_1 + W_2 \times C_2 + W_3 \times C_3 + W_4 \times C_4 + W_5 \times C_5 \]

Where

\[
\begin{align*}
C_1 &= \text{# of Content coupling incidents}, \\
C_2 &= \text{# of Common coupling incidents}, \\
C_3 &= \text{# of Control coupling incidents}, \\
C_4 &= \text{# of Stamp coupling incidents}, \\
C_5 &= \text{# of Data coupling incidents}
\end{align*}
\]

and

\[
\begin{align*}
W_1 &= 3; \\
W_2 &= 2.5; \\
W_3 &= 2; \\
W_4 &= 1.5; \\
W_5 &= 1.0.
\end{align*}
\]

5

Note that \( \sum_{i=1}^{5} W_i = 10 \)

The reason for choosing the weights will be discussed in the metric weights section as it requires discussion of another metric, DVC, proposed below.

In simpler term coupling is directly related to vulnerability and vulnerability is inversely related to software security i.e.

\[ \text{Coupling} \equiv \text{Vulnerability} \equiv \frac{1}{\text{Security}} \]
Since AVC metric incorporates all the major types of software coupling with weight factor, showing correlation between AVC metric and vulnerability bug report will show the thesis hypothesis true.

**Degree of Vulnerability in terms of Coupling Metric (DVC)**

Next we provide a slightly improved version of the AVC metric, called DVC. Degree of Vulnerability via Coupling, DVC metric is defined as:

\[
\text{DVC} = 1 - \left( \frac{1}{\text{AVC}} \right), \text{ where } 0 < \text{DVC} < 1
\]

Where

\[
\text{DVC} = 0 \text{ implies possibly low or no coupling and possibly less vulnerable}
\]

\[
\text{DVC} = 1 \text{ implies possibly high coupling and possibly highly vulnerable}
\]

DVC is simple, and it is used to capture the degree of vulnerability with respect to coupling. Since it is bounded by 0 and 1 it is more elegant and easier to comprehend. If we could prove our hypothesis true using AVC metric then DVC metric can be used to show how vulnerable a component/software is.

**Metric Weights**

The weights for AVC metric are chosen based on the coupling type preference. For example, content coupling is the least preferred and it is given more weight. Our claim is that coupling is directly proportional to vulnerability hence the least a coupling type is preferred the more the weights given which in turn will have a larger impact in vulnerability.
To start with the data coupling is given a weight of 1 as it the most preferred coupling type. The reason for choosing 1 as the weight for data coupling is that if only data coupling is present in a software with just one incident then the AVC metric will become 1 and the corresponding DVC will become 0. With DVC zero the software may be considered as less vulnerable. It is also logically evident that having just one data coupling incident may not necessarily make the software vulnerable as data coupling is one of the most preferred coupling types.

We do not want to choose any random weights for the rest of the coupling types and wanted to have equal weights interval and chose 0.5. This is because with 0.5 weights interval, the weight for the control coupling (center pivot in the preferred coupling types) become 2 and just having one instance of control coupling with the rest of coupling incidents zero makes the AVC metric 2 and DVC metric 0.5. With DVC being 0.5 the software can be considered either vulnerable or non-vulnerable. It makes sense logically that just having one incident of control coupling, may or may not make the software vulnerable as control coupling is placed middle in the preferred coupling types. With 1 as weight for data coupling, 2 for control coupling and 0.5 as the interval the weights for content coupling becomes 3, common coupling becomes 2.5 and stamp coupling becomes 1.5. The sum of weights becomes 10. Note that we assumed the interval to be of equal distance of .5. While it is possible that the coupling differences are not equidistance away, currently there is no coupling metric that shows the interval
differences. Thus we made a simplistic assumption here and used the same .5 in weight differences.

Although pragmatically there cannot be software written without coupling but theoretically it is possible. Under this circumstance the AVC metric can be considered to be equivalent to having one incident of data coupling which makes the AVC metric to be 1. With AVC metric being 1 DVC metric becomes 0 and intuitively this can be considered as no coupling.

Changing the weights will affect the AVC metric only when it comes to interval or ratio scales. Weights change will not make much of difference as long as the order of the coupling types is preserved. Since the coupling types order is important i.e. ordinal scale and we preserve it, changing the weights will not affect the metric. With our assumption of using the same .5 as the distance between the weights, we have also moved the metric up a scale to interval level. This allows us to perform arithmetic operation of addition and subtraction on our AVC metric. The weights are chosen to make the metric logically convincing.

**Metric Value Interpretation**

In this section we will talk about the DVC metric value interpretation. Based on various possible scenarios we will propose a value for DVC metric that can serve as a possible guiding post for vulnerability.
Case 1:

There is one instance of each coupling types

\[ C_1 = C_2 = C_3 = C_4 = C_5 = 1 \]

\[ W_1 = 3; W_2 = 2.5; W_3 = 2; W_4 = 1.5; W_5 = 1.0 \]

\[ AVC_1 = 10 \quad DVC_1 = 0.9 \]

Case 2:

There is one instance of Content and Common Coupling. Zero instances of Control, Stamp and Data Coupling

\[ C_1 = C_2 = 1; C_3 = C_4 = C_5 = 0 \]

\[ W_1 = 3; W_2 = 2.5; W_3 = 2; W_4 = 1.5; W_5 = 1.0 \]

\[ AVC_2 = 5.5 \quad DVC_2 = 0.82 \]

Case 3:

There is one instance of Control, Stamp and Data Coupling. Zero instances of Content and Common Coupling

\[ C_1 = C_2 = 0; C_3 = C_4 = C_5 = 1 \]

\[ W_1 = 3; W_2 = 2.5; W_3 = 2; W_4 = 1.5; W_5 = 1.0 \]

\[ AVC_3 = 4.5 \quad DVC_3 = 0.78 \]

Case 4:

There is one instance of Content, Common and Control Coupling. Zero instances of Stamp and Data Coupling
C1 = C2 = C3 = 1; C4 = C5 = 0
W1 = 3; W2 = 2.5; W3=2; W4=1.5; W5=1.0
AVC4 = 7.5 DVC4 = 0.86

Case 5:
There is one instance of Stamp and Data Coupling. Zero instances of Content, Common and Control Coupling
C1 = C2 = C3 = 0; C4 = C5 = 1
W1 = 3; W2 = 2.5; W3=2; W4=1.5; W5=1.0
AVC5 = 2.5 DVC5 = 0.6

Case 6:
There is one instance of Data Coupling. Zero instances of Content, Common, Control and Stamp Coupling
C1 = C2 = C3 = C4 =0; C5 = 1
W1 = 3; W2 = 2.5; W3=2; W4=1.5; W5=1.0
AVC6 = 1.0 DVC6 = 0

Case 7:
There is one instance of Content Coupling. Zero instances of Common, Control, Stamp and Data Coupling
C1 = 1; C2 = C3 = C4 = C5 = 0
W1 = 3; W2 = 2.5; W3=2; W4=1.5; W5=1.0
AVC7 = 3 DVC7 = 0.67

Case 8:
There is one instance of Common Coupling. Zero instances of Content, Control, Stamp and Data Coupling
C2 = 1; C1 = C3 = C4 = C5 = 0
W1 = 3; W2 = 2.5; W3=2; W4=1.5; W5=1.0
AVC8 = 2.5 DVC8 = 0.6

Case 9:
There is one instance of Control Coupling. Zero instances of Content, Common, Stamp and Data Coupling
C3 = 1; C1 = C2 = C4 = C5 = 0
W1 = 3; W2 = 2.5; W3=2; W4=1.5; W5=1.0
AVC9 = 2 DVC9 = 0.5

Case 10:
There is one instance of Stamp Coupling. Zero instances of Content, Common, Control and Data Coupling
C4 = 1; C1 = C2 = C3 = C5 = 0
W1 = 3; W2 = 2.5; W3=2; W4=1.5; W5=1.0
AVC10 = 1.5 DVC10 = 0.33
Case 11:
All instances of coupling are zero. This can be considered as a coupling with one instance of data coupling.
\[ C_1 = C_2 = C_3 = C_4 = C_5 = 0; \]
\[ W_1 = 3; W_2 = 2.5; W_3 = 2; W_4 = 1.5; W_5 = 1.0 \]
\[ AVC_{11} = 1 \quad DVC_{11} = 0; \]

Case 12:
There is one instance of Common, Stamp and Data Coupling. Zero instances of Content and Control Coupling.
\[ C_1 = C_3 = 0; C_2 = C_4 = C_5 = 1; \]
\[ W_1 = 3; W_2 = 2.5; W_3 = 2; W_4 = 1.5; W_5 = 1.0 \]
\[ AVC_{12} = 5 \quad DVC_{12} = .80 \]

Case 13:
There is one instance of Common and Data Coupling. Zero instances of Content, Control and Stamp Coupling.
\[ C_2 = C_5 = 1; C_1 = C_3 = C_4 = 0; \]
\[ W_1 = 3; W_2 = 2.5; W_3 = 2; W_4 = 1.5; W_5 = 1.0 \]
\[ AVC_{13} = 3.5 \quad DVC_{13} = .71 \]
Case 14:
There is one instance of Control and Stamp Coupling. Zero instances of Content, Common and Data Coupling.
C3 = C4 = 1; C1 = C2 = C5 = 0;
W1 = 3; W2 = 2.5; W3 = 2; W4 = 1.5; W5 = 1.0
AVC14 = 3.5 DVC14 = .71

Case 15:
There is one instance of Common and Control Coupling. Zero instances of Content, Stamp and Data Coupling.
C2 = C3 = 1; C1 = C4 = C5 = 0;
W1 = 3; W2 = 2.5; W3 = 2; W4 = 1.5; W5 = 1.0
AVC15 = 4.5 DVC15 = .78

Case 16:
There is one instance of Content and Data Coupling. Zero instances of Common, Control and Stamp Coupling.
C1 = C5 = 1; C2 = C3 = C4 = 0;
W1 = 3; W2 = 2.5; W3 = 2; W4 = 1.5; W5 = 1.0
AVC16 = 4.0 DVC16 = .75
Case 17:
There is one instance of Content and Stamp Coupling. Zero instances of Common, Control and Data Coupling.
C1 = C4 = 1; C2 = C3 = C5 = 0;
W1 = 3; W2 = 2.5; W3=2; W4=1.5; W5=1.0
AVC17 = 4.5 DVC17 = .78

Case 18:
There is one instance of Content and Control Coupling. Zero instances of Common, Stamp and Data Coupling
C1 = C3 = 1; C2 = C4 = C5 = 0;
W1 = 3; W2 = 2.5; W3=2; W4=1.5; W5=1.0
AVC18 = 5.0 DVC18 = .8

Case 19:
There is one instance of Content and Common Coupling. Zero instances of Control, Stamp and Data Coupling
C1 = C2 = 1; C3 = C4 = C5 = 0;
W1 = 3; W2 = 2.5; W3=2; W4=1.5; W5=1.0
AVC19 = 5.5 DVC19 = .82
Case 20:

There is one instance of Content, Common, Control, and Stamp Coupling. Zero instances of Data Coupling.

C1 = C2 = C3 = C4 = 1; C5 = 0;
W1 = 3; W2 = 2.5; W3=2; W4=1.5; W5=1.0
AVC20 = 9 DVC20 = .89

Case 21:

There is one instance of Content, Control, Stamp and Data Coupling. Zero instances of Common Coupling.

C1 = C5 = C3 = C4 = 1; C2 = 0;
W1 = 3; W2 = 2.5; W3=2; W4=1.5; W5=1.0
AVC21 = 7.5 DVC21 = .87

Case 22:

There is one instance of Content, Common, Stamp and Data Coupling. Zero instances of Control Coupling.

C1 = C2 = C5 = C4 = 1; C3 = 0;
W1 = 3; W2 = 2.5; W3=2; W4=1.5; W5=1.0
AVC22 = 8 DVC22 = .88
Case 23:

There is one instance of Content, Common, Control and Data Coupling. Zero instances of Stamp Coupling.

C1 = C2 = C3 = C5 = 1; C4 = 0;
W1 = 3; W2 = 2.5; W3=2; W4=1.5; W5=1.0
AVC23 = 8.5 DVC23 = .88

Case 24:

There is one instances of Common, Control, Stamp and Data Coupling. Zero instances of Content Coupling

C5 = C2 = C3 = C4 = 1; C1 = 0;
W1 = 3; W2 = 2.5; W3=2; W4=1.5; W5=1.0
AVC24 = 7 DVC24 = .86

Case 25:

There is one instance of Common, Control and Stamp Coupling. Zero instances of Content and Data Coupling

C1 = C5 = 0; C3 = C4 = C2 = 1;
W1 = 3; W2 = 2.5; W3=2; W4=1.5; W5=1.0
AVC25 = 6 DVC25 = .83
Case 26:
There is one instance of Content, Control and Stamp Coupling. Zero instances of Common and Data Coupling.
C2 = C5 = 0; C3 = C4 = C1 = 1;
W1 = 3; W2 = 2.5; W3=2; W4=1.5; W5=1.0
AVC26 = 6.5 DVC26 = .85

Case 27:
There is one instance of Content, Stamp and Common coupling. Zero instances of Control and Data Coupling.
C3 = C5 = 0; C1 = C4 = C2 = 1;
W1 = 3; W2 = 2.5; W3=2; W4=1.5; W5=1.0
AVC27 = 7 DVC27 = .86

Case 28:
There is one instance of Content, Common and Data coupling. Zero instances of Control and Stamp Coupling.
C3 = C4 = 0; C1 = C2 = C5 = 1;
W1 = 3; W2 = 2.5; W3=2; W4=1.5; W5=1.0
AVC28 = 6.5 DVC28 = .85
**Case 29:**
There is one instance of Content, Stamp and Data Coupling. Zero instances of Common and Control Coupling.

\[
C2 = C3 = 0; C1 = C4 = C5 = 1;
\]
\[
W1 = 3; W2 = 2.5; W3=2; W4=1.5; W5=1.0
\]

\[
AVC29 = 5.5 \quad DVC29 = .82
\]

**Case 30:**
There is one instance of Content, Control and Data Coupling. Zero instances of Common and Stamp Coupling

\[
C2 = C4 = 0; C1 = C3 = C5 = 1;
\]
\[
W1 = 3; W2 = 2.5; W3=2; W4=1.5; W5=1.0
\]

\[
AVC30 = 6 \quad DVC30 = .83
\]

**Case 31:**
There is one instance of Common, Stamp and Data Coupling. Zero instances of Content and Control Coupling

\[
C1 = C3 = 0; C2 = C4 = C5 = 1;
\]
\[
W1 = 3; W2 = 2.5; W3=2; W4=1.5; W5=1.0
\]

\[
AVC31 = 5 \quad DVC31 = .8
\]
Case 32:

There is one instance of Control and Data Coupling. Zero instances of Content, Common and Stamp Coupling

\[ C3 = C5 = 1; \quad C1 = C2 = C4 = 0; \]

\[ W1 = 3; \quad W2 = 2.5; \quad W3=2; \quad W4=1.5; \quad W5=1.0 \]

AVC32 = 3 DVC32 = .67

DVC Average of all the cases

\[
\text{DVCAvg} = \frac{\sum_{i=1}^{32} \text{DVCI} / \text{Number of Cases}}{32}
\]

\[
\text{DVCAvg} = \frac{0.9 + 0.82 + 0.78 + 0.86 + 0.6 + 0 + 0.67 + 0.6 + 0.5 + 0.33 + 0 + 0.8 + 0.71 + 0.71 + 0.78 + 0.75 + 0.78 + 0.8 + 0.82 + 0.89 + 0.87 + 0.88 + 0.88 + 0.86+ 0.83 + 0.85 + 0.86 + 0.85 + 0.82 + .83}{32} = 23.1 / 32
\]

\[ = 0.72 \]

DVC Metric Value greater than or equal to 0.72 can be considered as highly Vulnerable and less than 0.72 can be considered as less vulnerable. The DVC metric value of 0.72 may be used as a reference number for future research into using coupling as a predictor for security vulnerability.

Vulnerability Data Collection

We used Neuhaus, et al [1] data findings as inputs for our metrics in this thesis. Neuhaus and team have used Mozilla source code and database for their data collection.
as all the relevant data are available freely. Vulnerabilities are announced in security advisories that provide users workarounds or pointers to fixed versions and help them avoid security problems. In Mozilla, advisories also refer to a bug report in the Bugzilla (Mozilla bug tracker) database. This information has been used by Neuhaus, et al to map vulnerabilities to components through the fixes that remove the defect. They retrieved all advisories from the Mozilla’s using “Known Vulnerabilities in Mozilla Products” page. Each advisory called Mozilla Foundation Security Advisory (MFSA) will point to one or more bugs in the bugzilla. MFSA will look like the one in the figure 4.1

**Figure 4.1**
The link at the bottom of the MFSA in the References section will provide details on the bug.

https://bugzilla.mozilla.org/show_bug.cgi?id=362213

The id at the end of the link corresponds to the bug identifier of the defect that caused the vulnerability. These bug identifiers are all collected and used to identify the corresponding fixes in the version archive. The version archive will look like the one in figure 4.2. In the version archive of the sources each fix will have the bug identifier and once the fix has been identified, it is easy to find out the components that are affected.

**Figure 4.2**

Stephen Neuhaus, etal evaluated Vulture on the C++ code base of Mozilla and found that Vulture correctly identifies about two thirds of all vulnerable components. In
this work they have listed the top ten most vulnerable components in Mozilla with respect to vulnerable bug reports. In this thesis we have used this data of most vulnerable components together with their bug counts to devise our security metrics which in turn is used to associate with coupling metrics.

We directly obtained the raw data from Neuhaus’s group for our research purpose. We will use the Vulnerability bug report by Neuhaus for establishing our empirical data evidence in Chapter 5.

**Coupling Metrics Collection**

Understand C++ [24] a Statistical analysis tool is used in this thesis to collect Coupling Metrics. The tool is capable of generating several metrics like global functions, global variables, local variables, local functions, LOC, Comments, Ratio Comment to Code, complexity metrics, class OO metrics, etc. We are interested only in coupling metrics and could find few coupling types that we can use for our empirical study. These coupling types are global functions, global variables, and Module dependency (Fan in / Fan out) counts. We planned to use global functions and global variables together as one type called global count as a Common Coupling representative and plan to use module dependency count as a Content Coupling representative.

Currently our empirical data collection will be limited to Content and Common Coupling as these are the least preferred Coupling types and will not take into account of
other coupling types. Since content and common coupling are the least preferred coupling types and establishing evidence that the existence of these coupling types will affect the security may suffice to prove our hypothesis true.
Chapter 5

EMPIRICAL STUDY

Data Analysis and Evaluation

In this chapter we will use the raw data obtained from Dr. Stephen Neuhaus [1] et al, together with our calculated AVC and DVC metric to show empirical evidence to claim that our hypothesis is valid. Table 5.1 captures the top 10 most vulnerable components ordered by their bug count. The components refer to the modules in the Mozilla source code and it corresponds to a C or C++ source file.

As we discussed in Chapter 3 Mozilla security advisories come in as Mozilla Foundation Security Advisories (MFSA) with each MFSA could have more than one bugs associated. Table 5.1 has the most vulnerable components together with their bug count and MFSA. The bug count for each component in the table is the total number of security related bugs posted and fixed for that component. In this thesis we consider security bug count as a factor than MFSA and the bug count will be used to test our hypothesis. Bug severity is not captured in the bug count as we want to keep our data analysis simple.
<table>
<thead>
<tr>
<th>Bug Rank</th>
<th>Component</th>
<th>Bug Count</th>
<th>MFSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>jsobj.c</td>
<td>24</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>nsCSSFrameConstructor.cpp</td>
<td>17</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>jsfun.c</td>
<td>15</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>nsScriptSecurityManager.cpp</td>
<td>15</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>NsGlobalWindow.cpp</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>jcsript.c</td>
<td>14</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>jsiinterp.c</td>
<td>14</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>nsDOMClassinfo.cpp</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>nsGenericElement.cpp</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>nsDOCShell.cpp</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

The Table 5.2 below captures AVC metric calculation. As discussed in the chapters 2 and 4, Global count is considered as a representative of Common Coupling which was assigned the weight 2.5 and Dependency count is considered as a representative of Content Coupling which was assigned the weight 3.0. In our empirical
study we are considering only the two worst types of Coupling, Content and Common, and the rest of coupling types are not considered. AVC metric is calculated considering these two coupling types as the scope is limited because of lack of information. We thought that we would perform the empirical study by including just the two worst coupling types and if we can prove our hypothesis it would suffice. Future work needs to include all coupling types and include a deeper look at the bug severity and bug types, not just the count.

Table 5.2 AVC metric

<table>
<thead>
<tr>
<th>Component</th>
<th>Total Global Count (GC)</th>
<th>Weight W2</th>
<th>Total Dependency Count (DC)</th>
<th>Weight W1</th>
<th>AVC = GC * W1 + DC * W2</th>
</tr>
</thead>
<tbody>
<tr>
<td>jsobj.c</td>
<td>83</td>
<td>2.5</td>
<td>44</td>
<td>3.0</td>
<td>339.5</td>
</tr>
<tr>
<td>nsCSSFrameConstructor.cpp</td>
<td>6</td>
<td>2.5</td>
<td>15</td>
<td>3.0</td>
<td>60</td>
</tr>
<tr>
<td>jsfun.c</td>
<td>23</td>
<td>2.5</td>
<td>31</td>
<td>3.0</td>
<td>150.5</td>
</tr>
<tr>
<td>nsScriptSecurityManager.cpp</td>
<td>7</td>
<td>2.5</td>
<td>6</td>
<td>3.0</td>
<td>35.5</td>
</tr>
<tr>
<td>NsGlobalWindow.cpp</td>
<td>13</td>
<td>2.5</td>
<td>28</td>
<td>3.0</td>
<td>116.5</td>
</tr>
<tr>
<td>jcsript.c</td>
<td>23</td>
<td>2.5</td>
<td>36</td>
<td>3.0</td>
<td>165.5</td>
</tr>
<tr>
<td>Jsinterp.c</td>
<td>21</td>
<td>2.5</td>
<td>37</td>
<td>3.0</td>
<td>163.5</td>
</tr>
<tr>
<td>nsDOMClassinfo.cpp</td>
<td>21</td>
<td>2.5</td>
<td>24</td>
<td>3.0</td>
<td>124.5</td>
</tr>
<tr>
<td>nsGenericElement.cpp</td>
<td>31</td>
<td>2.5</td>
<td>18</td>
<td>3.0</td>
<td>131.5</td>
</tr>
<tr>
<td>nsDOCSHELL.cpp</td>
<td>11</td>
<td>2.5</td>
<td>10</td>
<td>3.0</td>
<td>57.5</td>
</tr>
</tbody>
</table>
Correlation is a measure of the degree of association between two variables, and we will be using correlation to “test” the strength of association between i) bug counts and AVC and ii) bug counts and DVC.

Pearson and Spearman Rank correlation coefficient are the two widely used statistics to measure the strength of correlation between two variables. Pearson is used when the data are normally distributed and are in interval or ratio scale. Spearman’s correlation is used when the data are in ordinal or nominal scale and it does not consider the magnitude of the data. In this thesis we would like to calculate both Pearson and Spearman rank correlation coefficients and rationale our findings.

**Pearson Correlation Coefficient**

\[
\text{Pearson Correlation Coefficient } = \frac{N \sum XY - (\sum X)(\sum Y)}{\sqrt{[N(\sum X^2 - (\sum X)^2)][N(\sum Y^2 - (\sum Y)^2)]}}
\]

- \(N\) = Number of values or elements
- \(X\) = Elements of First Variable
- \(Y\) = Elements of Second Variable
- \(\Sigma XY\) = Sum of the product of elements of First and Second Variables
- \(\Sigma X\) = Sum of First Variables
- \(\Sigma Y\) = Sum of Second Variables
- \(\Sigma X^2\) = Sum of square First Variables
- \(\Sigma Y^2\) = Sum of square Second Variables

**Spearman Correlation Coefficient**

\[
\text{Spearman Correlation Coefficient is } = 1 - \frac{6(\sum D^2)}{N(N^2 - 1)}
\]

\(D\) is the difference between the ranks of the two variables.
6 is a constant (it is always used in the formula),

\[ D = \text{difference between a subjects ranks on the two variables}, \]

\[ N = \text{number of subjects}. \]

According to Yonghee and Laurie [39], Correlation coefficient less than 0.3 means weak correlation, 0.3 to 0.5 means medium correlation and greater than 0.5 means strong correlation. However, the interpretation depends on the context of the usage. We use the reference values by Yonghee and Laurie in this thesis.

The Table 5.3 below includes the AVC metric for top 10 most vulnerable components of Mozilla, each component’s vulnerability bug count from Stephen Neuhaus [1] and the calculated Pearson Correlation and Spearman Rank Coefficient values.
Table 5.3 AVC metric and Vulnerability bug Correlation

<table>
<thead>
<tr>
<th>Component</th>
<th>AVC</th>
<th>Bug Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>jsobj.c</td>
<td>339.5</td>
<td>24</td>
</tr>
<tr>
<td>nsCSSFrameConstructor.cpp</td>
<td>60</td>
<td>17</td>
</tr>
<tr>
<td>jsfun.c</td>
<td>150.5</td>
<td>15</td>
</tr>
<tr>
<td>nsScriptSecurityManager.cpp</td>
<td>35.5</td>
<td>15</td>
</tr>
<tr>
<td>NsGlobalWindow.cpp</td>
<td>116.5</td>
<td>14</td>
</tr>
<tr>
<td>jscscript.c</td>
<td>165.5</td>
<td>14</td>
</tr>
<tr>
<td>jsinterp.c</td>
<td>163.5</td>
<td>14</td>
</tr>
<tr>
<td>nsDOMClassinfo.cpp</td>
<td>124.5</td>
<td>10</td>
</tr>
<tr>
<td>nsGenericElement.cpp</td>
<td>131.5</td>
<td>10</td>
</tr>
<tr>
<td>nsDOCShell.cpp</td>
<td>57.5</td>
<td>9</td>
</tr>
</tbody>
</table>

**Pearson Correlation** 0.66

**Spearman Rank Correlation** 0.235
As discussed DVC metric is calculated from AVC metric using the formula

\[ DVC = 1 - \left(1 / AVC\right) \]

The values of DVC metric for each of the most vulnerable components in Mozilla and their bug vulnerability bug count is shown in Table 5.4

**Table 5.4 DVC metric and Vulnerability bug correlation**

<table>
<thead>
<tr>
<th>Component</th>
<th>DVC</th>
<th>Bug Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>jsobj.c</td>
<td>0.997</td>
<td>24</td>
</tr>
<tr>
<td>nsCSSFrameConstructor.cpp</td>
<td>0.983</td>
<td>17</td>
</tr>
<tr>
<td>jsfun.c</td>
<td>0.993</td>
<td>15</td>
</tr>
<tr>
<td>nsScriptSecurityManager.cpp</td>
<td>0.972</td>
<td>15</td>
</tr>
<tr>
<td>NsGlobalWindow.cpp</td>
<td>0.991</td>
<td>14</td>
</tr>
<tr>
<td>jscsript.c</td>
<td>0.994</td>
<td>14</td>
</tr>
<tr>
<td>jsinterp.c</td>
<td>0.994</td>
<td>14</td>
</tr>
<tr>
<td>nsDOMClassinfo.cpp</td>
<td>0.992</td>
<td>10</td>
</tr>
<tr>
<td>nsGenericElement.cpp</td>
<td>0.992</td>
<td>10</td>
</tr>
<tr>
<td>nsDOCSHelli.cpp</td>
<td>0.982</td>
<td>9</td>
</tr>
</tbody>
</table>
Results

From our discussion earlier, Correlation Coefficient value greater than 0.5 means strong correlation. The Pearson Correlation Coefficient of 0.66 calculated in table 5.3 shows a strong correlation between AVC metric and Mozilla Vulnerabilities of top 10 vulnerable components. Also the correlation coefficient implies strong relation between Vulnerability and Coupling, as AVC metric is expressed in terms of coupling metrics.

Pearson correlation coefficient shows a high association between the existence of the worst two types of coupling in a software component and the number of bugs in that component. On the other hand, the Spearman rank correlation of 0.235 calculated in table 5.3 implies that the ranking of the components by number of bugs and the ranking of components by the number of two worst types of coupling do not show high association. Spearman rank correlation shows weak correlation between the AVC metric and Mozilla vulnerabilities for top 10 vulnerable components.

The following reasons can be attributed to the low spearman rank correlation.

a. We do not include the types of bug in our empirical study. Probably if we had included the types of bugs and their severity and then ranked accordingly by considering only more severe bugs, we could have shown high spearman rank correlation.

b. Also we considered only top 10 most vulnerable components of Mozilla. Had we included more vulnerable components, not just top 10, probably we could have shown high spearman rank correlation.
From the Pearson and Spearman Rank correlation we can infer that the high level picture of coupling and amount of bugs may be associated, but the detailed ranking may not. This inference makes our hypothesis still valid and true.

The DVC metric values are also pretty high > 0.9 for each of the vulnerable components. Since DVC metric is nothing but the degree of vulnerability expressed in terms of coupling, high DVC metric implies high coupling in the vulnerable components of Mozilla.

From the correlation analysis, we believe that our posed hypothesis: “Software Coupling is a factor influencing Software Security Vulnerability” is valid.
Chapter 6

CONCLUSION

In this thesis we initially laid theoretical foundation by providing definition and explanation of Software Security and Software Coupling. We then explained key security concepts like threats, attackability, Damages and Vulnerability and the relationship between them. Later we explained that Security and Vulnerability are inversely related. We also listed out the different coupling types Content, Common, Control, Stamp and Data Coupling and gave their definitions from our references.

We then touched upon software metrics and measurement framework to lay down the foundation for our metrics. We discussed about common vulnerabilities in software and their possible relationship to Software Coupling. Later we put forth our hypothesis that “Software Coupling is a factor influencing Software Vulnerability” and discussed about our empirical study methodology. In our methodology we devised two metrics namely AVC and DVC.

AVC metric is defined using the traditional five types of coupling together with the weights based on the individual coupling preferences. We gave rationale for choosing different weights and weights interval. DVC metric is an improved version of AVC metric and it captured the degree of vulnerability with respect to coupling. We then
proposed a reference value for DVC metric of 0.72 that can be used to classify vulnerability. We later discussed about our empirical data collection.

In this thesis we are able to show empirical data evidence for correlation between coupling, Content and Common coupling, and Software Vulnerability. Using the AVC/DVC metrics proposed we are able to prove our hypothesis that **Software Coupling is a factor influencing Software Vulnerability**. Although the detailed ranking of bugs and AVC metric shows weak relation by Spearman Rank Correlation, the high level picture of coupling and amount of bugs is strongly associated by Pearson Correlation. This holds our argument valid and our hypothesis true.

Furthermore, the DVC metrics can be used for prediction of vulnerability as higher values of DVC metric mean high coupling which in turn might expose software vulnerability. However this needs to be validated by applying this metric in other software projects to claim that software coupling influences software security in general.

This thesis incorporates only two types of coupling, least preferred coupling types, in calculating the AVC metric and it would have been better to include other coupling types. Further statistical analysis of Mozilla is needed to come up with metrics that can include other coupling types. By doing so, the empirical analysis can be more accurate. Future work needs to include all coupling types and include a deeper look at the bug severity and bug types, not just the count. By doing so possibly the Spearman Rank Correlation may show strong association too. Moreover, the correlation can just show
only association but not cause-and-effect relationship. Thus future work needs to delve more into the cause-and-effect relationship.

The thesis study can be extended to other software projects to corroborate the hypothesis. There is no doubt that software vulnerability might be affected by other internal attributes like complexity, cohesion, etc. Our thesis approach can be used to study if these internal attributes can affect software vulnerability.

Authors Michael Yanguo Liu [5] [23] etal, have shown empirical data evidence for association between Software Coupling and individual Vulnerabilities like DoS and URL Jumping. In this thesis we have tried to show relation between Software Coupling and Vulnerability in general and attained some success too. These studies support the notion that software coupling is a fertile area to focus on to close out potential security vulnerabilities.
References

[1] Stephan Neuhaus, Thomas Zimmermann, Andreas Zeller, *Predicting vulnerable Software components*, Saarland University, Saarbrücken, Germany


[15] Ben Smith Yonghee Shin Laurie Williams, Proposing SQL Statement Coverage Metrics, Department of Computer Science, North Carolina State University


[18] Frank Kargl, Joern Maier, Michael Weber, Protecting Web Servers from Distributed Denial of Service Attacks, Department of Multimedia, Computing University of Ulm, Germany


[23] Yanguo (Michael) Liu, Issa Traore, *Complexity Measures for Secure Service-Oriented Software Architectures*, Department of Electrical and Computer Engineering, University of Victoria, Canada


