1. INTRODUCTION

Software Engineering is the application of systematic, disciplined and quantifiable approach to the design, development, operation and maintenance of software. There has been no consensus on the exact definition of the term “Software Engineering”. The term “Software Engineering” first appeared in 1968 and was predominantly aimed at tackling the problem then described as “Software Crisis”.

In the early decades of the software industry, the engineering of “quality” software was a big challenge to the software developers and managers of the period. Many sound principles that aid in the development of quality software were proposed and the adoption of these principles lead to a perceivable improvement in the quality of the software being engineered. Such principles related both to the product and the process employed to engineer the product. Examples of such principles include reviews, inspections, and Software Quality Assurance.

1.1 Metrics in Software Engineering

The importance of measurement in any engineering discipline cannot be overstated. Software Engineering, being an engineering discipline, can be no exception to this. Pressman quotes Lord Kelvin – “When you can measure what you are speaking about and express it in numbers, you know something about it; but when you cannot measure, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind” – to highlight the importance of measurement (Pressman, 2001). Software Metrics refers to a broad range of measurements for Computer Software.

• To Increase Understanding: Measurement greatly aids in the understanding of Software and Software Engineering Processes leading to better management of Software Projects and improvements in the Software Engineering Processes. For example, if an organization has collected measures of process and product characteristics such as number of defects found in a review, size of the product, effort required to develop the product, it can uncover novel relationships among the various measures such as the effort required to develop a software being dependent on size.

• To aid in the management of Software Projects: Knowledge gained about Software Processes can be used to estimate elements like effort and track results against estimates.

• To Guide improvements in Software Engineering Processes: Software Measurement is a key part in any process improvement programme. Knowing the quality of the product developed using both the original and the new processes is vital to ascertain the effectiveness of the process improvement strategy.

The Software Measurement Guide Book also warns that the focus should be on applying the acquired results than on collecting data.

1.1.1 Software Metrics Definition

As with many terminologies in Software Engineering, Metrics has come to mean different things to different people. Pressman clarifies the distinction between the terms – measure, measurement and metrics. When a single data point has been collected, as, for example, the number of errors uncovered in the review of a single module, a measure has been established. Measurement occurs as a result of collection of one or more data points. For example, for all the reviews conducted, the number of
errors uncovered in each review. A Metric relates the individual measures in some way. For example, the average number of errors found per review (Pressman, 2001). Essentially, software metrics deals with the measurement of the software product and the process by which it is developed (Mills, 1988). Here product refers not only to the final Software in the executable form but also the associated documentation and intermediate deliverables. Various measures are collected about both the products and the processes by which they are developed. These measures are then used to derive metrics that are expected to serve as guidelines for improving the quality of the products and the processes employed to engineer them.

1.1.2 The Need for Metrics

Mills observes that the software development scene is often characterized by schedule and cost estimates that are grossly inaccurate, software of poor quality, and a productivity rate that is increasing more slowly than the demand for software (Mills, 1988). This situation has often been referred to as “software crisis” (Arthur, 1985). Much of the situation can be attributed to ineffective software management that stems from poor measurement and lack of sound measures. Any improvement in the management process cannot be complete without an increased ability to identify, measure and control vital attributes of the development process. So the goal of Software Metrics is to address this challenge: identification and measurement of the essential parameters that affect software development (Mills, 1988).

Park, Goethert and Florac discuss the following reasons for measurement (Park, Goethert, & Florac, 1996):

i. To Characterize – This is done to gain understanding of processes, products and resources

ii. To Evaluate – This is done to determine status with respect to plans
iii. To Predict – This is done to uncover relationships among processes and products so that the values of some attributes can be used to predict the values of others.

iv. To Improve – Measures can help identify roadblocks, inefficiencies and uncover opportunities for improvement in quality.

1.1.3 The History of Software Metrics

Fenton and Neil claim that the history of software metrics dates back to mid 1960’s when Lines Of Code were used to measure programmer productivity and Effort (Fenton & Neil, n.d.). In this sense, the history of Metrics predates the History of Software Engineering. But advancements in Software Engineering have not seen a parallel in Software Metrics. Glass states that Software Engineering being an empirical discipline, Software Metrics must have come to play a pivotal role in it and yet Metrics continue to lie at the margins (Glass, 1994). Fenton and Neil attribute the lack of appreciation of the real value of metrics by the industry as the main reason behind this. Fenton and Neil also opine that even the recent upsurge in the industrial interest on Metrics is mainly to satisfy some external assessment body and substantiate their claim by the fact that the CMM has been the single major contributor for interest in Metrics in the US (Fenton & Neil, n.d.).

Many metrics have been proposed and used since 1960’s but Fenton and Neil state that the major rationale behind the development of the metrics was to predict or estimate effort, cost of processes or quality of the product (Fenton & Neil, n.d.). Various “size” measures were proposed for the same as it was assumed that size is the major driver for both cost and quality. The “Lines of Code (LOC)” metric was precisely used to measure the software size and continues to be used today to measure size as well as programmer productivity. LOC was also used to measure program
quality, though indirectly, through measures such as defects per KLOC. In the opinion of Fenton and Neil, the work of Akiyama (Akiyama, 1971) in proposing a model for module defect density in terms of KLOC was the first attempt to use metrics for quality prediction.

### 1.1.4 Characteristics of Ideal Metrics

Miller lists the following as the characteristics required of good Software Metrics (Miller, 1988). The metrics should be:

- Simple, and precisely definable, so that it is clear how the metric can be evaluated
- Objective, to the greatest extent possible
- Easily obtainable
- Valid – the metric should measure what it is intended to measure
- Robust – relatively insensitive to insignificant changes in the process or product.

The case for simplicity is that if the computation of the metric requires complex procedures it is likely that the procedure will be misunderstood and interpreted differently by different people. The most important characteristic is validity. If for example, a metric is defined for Software Size, it should be a valid indicator of the size. The metric need not be valid in absolute terms but should at least be valid in a relative context. For example, a metric for software size should give 2 times more value for a software that is 2 times as large as another. While all efforts are taken to achieve all these characteristics, the inherent complexity of the task of deriving sound metrics forces to make compromises as long as the compromises do not diminish the validity of the metrics.
1.1.5 Classification of Software Metrics

Miller classifies Metrics as Product Metrics that measure products such as source code and test case documentation and process metrics that measure the processes employed to engineer products such as number of defects found in a review (Miller, 1988). Metrics are also classified as subjective or objective depending upon whether these give identical values when computed by different people. Metrics can also be classified as primitive or computed on the basis of whether they can be directly observed or computed from other metrics. Grady classifies Metrics as private or team public or public (Grady, 1992). Private Metrics deal with data that are private and serve as indicators to the individual only. For example, the number of errors discovered in a module. Some metrics are private to a team but public to all the members of the team. For example, defects reported for major software functions. Public Metrics assimilate information that was originally private to individuals and teams. For example, project level defect rates absolutely not attributed to individuals.

1.1.6 Controversies about Software Metrics

Pressman states that the Software Engineering community has at last started to recognize the importance of measurements with a lot of frustration and more than a little controversy (Pressman, 2001). Software developers seem to persist in neglecting the importance of measurement. There is also a lot of confusion about what to measure and how the results are to be interpreted. Pressman attributes this state of affairs to cultural issues. Part of the Software Developers’ aversion to metrics can be attributed to the fact that metrics are perceived to be tools used for appraisal or threatening individual developers. To overcome this problem, Grady suggests a “Software Metrics Etiquette” that encompasses the following principles (Grady, 1992):
i) Provision of regular feedback to individuals and teams who collect metrics

ii) Refraining from using Metrics to appraise individuals

iii) Refraining from using Metrics to threaten individuals or teams

iv) Not considering those metrics that indicate a problem area as “negative”

v) Refraining from obsession on a single metric to the exclusion of all other metrics

The discipline of Software Engineering is not as matured as other disciplines and hence there are and there will continue to be a lot of controversies and differing opinions on many issues. Metrics will definitely be one among them. Over the years a lot of researchers in Software Engineering have questioned the utility of Metrics in Software Engineering. Most of the controversies seem to be around “quality metrics” which are expected to serve as quantitative indicators of Software Quality Characteristics. Pressman observes that criticisms of specific metrics are common but many of the critiques focus on esoteric issues and miss the primary objective of measurement – to help the engineer establish a systematic and objective way to gain insight into his or her work and to improve product quality as a result (Pressman, 2001).

1.2 Secure Software Engineering

With the penetration of software into many areas such as business, commerce and governance, security began to evolve as one of the major challenges confronting the software engineers. The development of distributed computing in general, and the World Wide Web in particular compounded the problem greatly, as software was no longer restricted to a single machine. Many early attempts to consider security as just another quality attribute expected of software (Chess, 2005), failed because such an approach was clearly insufficient to address the big threat posed by software security compromises.
There have been attempts to incorporate many principles originally proposed for ensuring software quality to the engineering of secure software. Littlewood et al. attempt to draw on the parallels between the challenges posed by Software Reliability and Software Security (Littlewood et al., n.d.). Many software engineering principles were extended to accommodate the new challenges presented by the security aspect. For Example, Software Quality Assurance is now complemented by Software Security Assurance. The Designer is now required to assess his design not only for its conformance to the explicitly stated functional requirements but also for security.

1.3 Research Issues pertaining to interactions between Software Engineering and Security

Devanbu and Stubblebine have highlighted the research issues that arise as a result of interactions between Software Engineering and Security roughly on the lines of the waterfall model – starting from requirements, proceeding through design and ending up with deployment and administration (Devanbu & Stubblebine, 2000).

1.3.1 Analysis

In the conventional method of software engineering, a principled approach is advocated for requirements gathering and elicitation. The requirements are gathered from the customer and a “Software Requirements Specification” is produced at the culmination of the analysis task. This specification is signed-off both by the software engineer and the customer and it becomes a “contract” for software development.

Various requirements are gathered during this phase but sadly, most SRS’s do not address the “Security” requirement properly. Security is a “non-functional” requirement and as developers grapple with the problem of getting the functionality right without overrunning schedules or budgets, security is not the utmost concern for system developers – even in systems where security threats can be easily perceptible.
(Doshi, 2002). Hence, Software Security is an “afterthought” i.e., security requirements can be considered seriously if the functional requirements are met and the project is within the schedule and budgets. As observed by Brooks, this is seldom, if ever, the case (Brooks, 1987).

In the current era of computing, as pointed out earlier, security is too important a requirement to be ignored or treated with a low-priority. The traditional requirements analysis focuses on requirements articulated by the customer directly. But, the security requirement may not be directly apparent to the customer and hence he may not articulate it. There is an imperative necessity to treat this “security” requirement as an “implicit” requirement – i.e. requirement that is so fundamental that the customer does not explicitly state it. If the customer, is unaware about the security requirement he needs to be educated on its necessity. If he is aware, he needs to be assured the developed software will meet his security requirements appropriately.

At a next level, it might be even wise to develop a separate “Software Security Specification”, that would detail the various aspects of security required by the software and would serve as a guide in the future for checking conformity of the developed software with the stated specifications. This specification needs to be given the same level of attention i.e. it needs to be reviewed, audited and maintained and available to both the developer and the customer, as the SRS.

1.3.2 Design

Conventional Software Design focuses on Data Design – which specifies “data structures” that will be used, Architectural Design – which defines the relationship between major structural elements of the software, Interface Design – which describes how the system communicates with systems that interoperate with it and Component-level Design – which transforms structural elements of the software architecture into a
procedural description of software components (Pressman, 2001). During design, the decisions taken ultimately affect the success of software construction and the ease with which software is maintained. Because design is a place where “quality” is fostered in software engineering, it is very imperative that the security aspect is addressed adequately in the design.

Every requirement in the “Software Security Specification” should be traceable to a “Security” Design element. The scope of Software design can be broadened to include “Security Design” along with the other 4 elements stated earlier. This Security Design would describe the security aspects of various structural elements in the software. For example, whenever a module is designed, the specification for the module should include a separate section for “Security” that would describe the security features of that module. Care needs to be taken to ensure that the security aspect of one module is not compromised when it interacts with another module. All this would necessitate a very big effort on the part of the software designer.

A possible research area for researchers in software engineering is the creation of new notations for describing security. Current notations do not address security directly and researchers can create new notations or enhance existing notations such as UML to include the security aspect.

1.3.3 Implementation

The importance of writing “quality” programs is now widely understood and appreciated as quality programs make testing and maintenance much easier. In the current era, it is not only important to write “quality” programs – programs have to be “secure”. What characterizes a “secure” source code? A possible answer is that a “secure” code is one which will resist any attacks made by malicious users. It is also
one which is free of any “security holes” that can possibly be exploited by hackers. With the umpteen tools available today, it is even possible to decompile a machine code to get the source code.

“Secure Software Coding” is an important paradigm that requires urgent attention. A possible research area for researchers in software engineering is the establishment of principles and guidelines for “Secure Software Coding”. One possibility is the development of source-code analysis tools that would throw some light on the security of programs given to it. These analyzers can look for patterns in the program that represent some known vulnerability and report them to the programmer. Programmers need to be trained to write “secure” programs and not simply programs that “meet the functional requirements”.

In addition to the normal inspections and reviews conducted for the source code, “security reviews and audits” should also be performed on the source code by highly trained security expert programmers to ensure “secure” source code. Every requirement specified in the “Software Security Specification” should be met by the source code.

Reuse of legacy code and Commercial-Off-The-Shelf (COTS) components presents an important challenge in this area. Legacy code generally does not contain ample provisions for security and when this is reused it is important to ensure that it is wrapped by appropriate security wrappers. These security wrappers have to be built for every COTS component and legacy code base used. In the context of COTS Components, for reasons pertaining to “intellectual property rights” most COTS vendors do not disclose the code for their components and this leaves little room for the organization to ensure that the component does not misbehave. Though various black-box and grey-box approaches have been proposed to resolve the issue, the
approaches are vulnerable to certain types of attacks and much more needs to be done in this regard.

The choice of programming language can also have a bearing on the security of the source code developed. For example languages like “Java” have more security features that are documented and available for all and the architecture of Java itself contributes to security to some level. But this does not mean that security need not be considered when developing programs in Java. Rather, the effort required for it can be reduced to some extent. When considering the programming language to use to develop the software the “security” dimension should also be considered.

1.3.4 Testing

In conventional software testing that begins after the source code is developed, the goal is to design a series of test cases that have a high likelihood of finding errors. Many testing tactics like “White Box Testing” and “Black Box Testing” are well documented and understood. But in the current scenario, there is a need for “Security Testing” as well. The objective of this testing would be to uncover security flaws and security holes in the program. As already stated, tools that would look for known insecure patterns in the program can be of vital help in this area.

“Security Testing” is a different activity that requires a different expertise than that required for conventional testing. This needs to be performed by highly trained security practitioners. The concept of “ethical” hackers – personnel who hack the software for good reasons – can be used and these “Ethical Hackers” should perform a thorough security testing.

A good research area for researchers is the development of principles and techniques for “Security Testing” and development of tools that can generate test
cases for the same. Current tools and testing principles are clearly inadequate for Security Testing.

In fact for most of the current systems “Penetration Testing” is the only means available for ensuring security. But as Chess argues, “The goal of the penetration testers is to find a small number of serious problems to justify their consulting fee and there’s no reason to believe that the penetration test revealed all of the problems in the application” (Chess, 2005). This clearly establishes the need for broader and versatile “Security Testing” practices.

1.3.5 Maintenance

The maintenance phase of software engineering begins after the software has been delivered to the customer. Conventionally, this includes Adaptation – which entails changing the software to meet new business requirements and policies, Enhancement – which enhances the software so that it meets new requirements not originally proposed, and Fixing – which entails the correction of bugs discovered as the software is used (Fairley, 2002). A reality with regard to security is that most of the “Security Flaws” are uncovered after delivery and they need to be fixed. This is especially true of Web applications which are hacked only after they are released. In fact one of the original goals of “Secure Software Engineering” is to minimize the frequency, potential and impact of such security flaws. But security flaws discovered post delivery should be fixed and it needs to be ensured that such fixes do not introduce new security flaws.

It is expected that, if the “Secure Software Engineering” principles outlined above are followed, the overhead incurred by the maintenance phase will be greatly reduced.
1.3.6 Configuration Management

According to Babich, Configuration management is the art of identifying, organizing, and controlling modifications to the software being built by a programming team (Babich, 1986). A common, if not oversimplified, view is that change is the enemy of security. Conventional techniques for configuration management should be augmented to incorporate the security aspect. Every change made, should be certified that it does not compensate the security of the software system. This, many times, may necessitate a redo of the “Security Testing” of the software system. But the overhead of doing the same is far better than the adverse consequences of not doing it.

1.4 SSE CMM

A lot of research has gone into Software Process Improvement with the underlying hope that better processes will lead to quality products. The Capability Maturity Model (CMM) proposed by the Software Engineering Institute (SEI) is aimed at providing a starting point for organizations interested in improving their processes or that do not have a institutionalized process at all. It can also be used as a benchmark for comparison and to aid understanding and to provide a framework for prioritizing actions for an organization that can learn from it’s past experiences. Zahran defines the Capability Maturity Model as “a structured framework for Software Process Assessment and Improvement” (Zahran, n.d.). A Maturity Level is a well defined evolutionary plateau on the path toward becoming a matured Software Organization. The model proposes 5 maturity levels and an organization can be appraised to be at any one of those levels based on it’s current practices. The 5 Levels are: i) Initial – where the Software Process is characterized as ad hoc, chaotic and success depends on individual effort, ii) Repeatable – where basic project
management processes are established and earlier successes can be repeated, iii) Defined – where all projects use an approved tailored version of the Organization’s Standard Software Process for development and maintenance, iv) Managed – where detailed measures of the process are collected and the process is quantitatively controlled, v) optimizing – where continuous process improvement is enabled by quantitative feedback from the process.

As the Software Engineering Community began to take cognizance of the increasing significance of Security, the Software Engineering Institute proposed the System Security Engineering – Capability Maturity Model (SSE CMM) akin to the CMM described above. This was done as it was realized that Security Engineering is a unique discipline requiring unique knowledge and skills. The SSE CMM provides a framework that can be used as way to measure and improve performance in the application of Security Engineering Principles (“SSE CMM Model Description Document”, 2003). The SSE-CMM addresses security engineering activities that span the entire life cycle of a secure system.

The SSE CMM has 2 dimensions – “domain” and “capability”. The Domain Dimension includes a set of “base practices”, while the capability dimension includes a set of “Generic Practices”. The major distinction between the 2 practices is that the “generic practices” indicate activities that have to be performed as a part of performance of base practices.

The SSE CMM contains 129 base practices out of which 61 are related to Security Engineering and 68 relate to project and organization domains. The 129 base practices are organized into 22 “Process Areas”. A Base Practice does not overlap with another and represents a best practice of the security community. It is applicable
using multiple methods. A Process Area puts together related base practices for ease of use. It can be improved as a distinct process.

The 22 Process areas are listed below:

- PA01 - Administer Security Controls
- PA02 - Assess Impact
- PA03 - Assess Security Risk
- PA04 - Assess Threat
- PA05 - Assess Vulnerability
- PA06 - Build Assurance Argument
- PA07 - Coordinate Security
- PA08 - Monitor Security Posture
- PA09 - Provide Security Input
- PA10 - Specify Security Needs
- PA11 - Verify and Validate Security

The remaining 11 process areas relate to the project and organization domains.

- PA12 - Ensure Quality
- PA13 - Manage Configuration
- PA14 - Manage Project Risk
- PA15 - Monitor and Control Technical Effort
- PA16 - Plan Technical Effort
- PA17 - Define Organization’s Systems Engineering Process
- PA18 - Improve Organization’s Systems Engineering Process
- PA19 - Manage Product Line Evolution
- PA20 - Manage Systems Engineering Support Environment
- PA21 - Provide Ongoing Skills and Knowledge
## PA22 - Coordinate with Suppliers

The Generic Practices pertain to the management aspects of a process and are applied to all processes. These generic practices are grouped into “Common Features” which are organized into “Capability Levels”. The Generic Practices are ordered according to maturity, unlike Base Practices that have no ordering.

The 5 capability Levels and their characteristics are listed below:

**Level 1:** Performed Informally – Focuses on whether an organization performs a process incorporating the base practices

**Level 2:** Planned and Tracked – Focuses on Project Level Definition, Planning and Performance Issues

**Level 3:** Well Defined – Focuses on disciplined tailoring from organization level defined processes

**Level 4:** Quantitatively Controlled – Focuses on Measurements

**Level 5:** Continuously Improving – Focuses on the use of gain accrued by the previous levels to explore ways of improvement.

The Common Features for each of the levels is listed below:

**Level 1:**

1.1 Base Practices are performed

**Level 2:**

2.1 Planning Performance
2.2 Disciplined Performance
2.3 Verifying Performance
2.4 Tracking Performance
Level 3:

3.1 Define a Standard Process
3.2 Perform the Standard Process
3.3 Coordinate the Process

Level 4:

4.1 Establishing Measurable Quality Goals
4.2 Objectively Managing Performance

Level 5:

5.1 Improving Organizational Capability
5.2 Improving Process Effectiveness

The SSE CMM Model Description Document cautions that it does not imply that Security Engineering should be carried out in isolation from other Engineering Disciplines and that the model promotes such integration taking the view that Security is pervasive across all disciplines - Software, Hardware and People (“SSE CMM Model Description Document”, 2003).

The formulation of such an elaborate CMM focused on Security shows that Security has received a lot of attention recently and that engineering of Secure Systems is becoming a great challenge to Developers and Organizations.

1.5 Software Security Measurement

While addressing the issue of engineering of quality software, developers and managers had a desperate need to know how much of quality is present in the software they are attempting to engineer so that they can tune their products and processes to ensure that they result in the required quality. The most easily graspable indicators were measures called metrics. For instance, if the developer has some
means to measure the reliability of the program he attempts to develop and sees that his program has 70% reliability, he can reassess his program and find some means to improve the reliability.

Some researchers believe that any attempt to quantify the quality of a software product or process is futile either because it is difficult or because it will lead to wrong conclusions. But many researchers agree that if the metrics are computed correctly in a given context, they will indeed be indicators of the attribute they attempt to measure. Voas et al. state that many metrics fail not because they are incorrect but because their value is often over sold (Voas et. al., 1996). As long as the meaning of metrics is taken in a relative context, there is nothing inherently wrong in using them.

Just as Metrics are important in ensuring the quality of the final product, it is imperative to measure the Security of software being engineered. Without measurement, Software Engineers have little control over Security resulting in a “less secure” system.

1.5.1 Current State of Practice for measuring Security

Chess highlights the current three flawed approaches for measuring Software Security (Chess, 2005):

- Penetration Testing as a Metric, where developers develop software with scant regard for security and subsequently a team of penetration testers is hired to find security related flaws in the Software. This approach is clearly insufficient as there is no guarantee that penetration testing has revealed all the problems and no feedback is provided to developers leaving little scope for developers being enlightened with security problems
• Measuring Software Security as a part of Software Quality suffers due to the one major difference between Security and other quality attributes. While most of the Quality Assurance activities verify a set of features against a specification, Security requires much more than well implemented security features (Chess, 2005).

• The feel good metric where organizations and individuals live in an illusion that if a software has not been hacked, it is secure, suffers from a major disadvantage – Security degrades as time passes. Novel and unconsidered attacks that exploit previously unknown vulnerabilities are on the rise.

1.5.2 Trailing Indicators

The Trustworthy Computing Security Development Lifecycle (Lipner & Howard, 2004) envisaged by Microsoft encompasses a broad range of activities from Risk Analysis, application of source code analysis tools, Security Testing and a final Security Review. Chess claims that this has led to a perceivable improvement in Security of Software released by Microsoft by presenting evidence based on the number of Security bulletins issued in the first 12 months of the release of Microsoft Windows 2000 and Microsoft Windows 2003 (Chess, 2005). This is indeed a very favorable trend toward Software Security Measurement but suffers from a major drawback – it is available only post delivery. Such indicators are described as “Trailing Indicators”.
1.6 Source Code Based Security Metrics

Chess throws light on derivation of Security Metrics based on Source Code (Chess, 2005). He claims that many vulnerabilities manifest themselves in code and that source code is the direct embodiment of software.

Source code analyzers that scan the code for previously known vulnerabilities can be developed and many of them are available. According to Chess, the three key attributes for good source code analysis are:

- **Accuracy** – The analyzer should accurately identify the vulnerability that is relevant to the application at hand. This becomes important because different applications are exposed to different threats and some vulnerabilities that are relevant for one application may be irrelevant for the other.

- **Precision** – The Source Code Analyzer must not produce false positives but at the same time point the analyst to manageable number of issues. The Naming Convention adopted by the source code analyzer is important meaning that the same issue should be given the same name even if the issue manifests slightly differently.
• Robustness – The ability to deal with large complex bodies of code is important for a Source Code Analyzer whose job is to analyze real world programs and not academic ones.

Chess also suggests there are 2 ways a source code analyzer can be used with regard to security.

• Developers use the source code analyzer regularly and improve their programs constantly while developing, or

• A Security Analysis team uses the source code analyzer for periodic review.

As the organization matures, it should resort more to the first approach.

1.7 Scope of the Research

Security Metrics can be computed at any stage of Software Engineering. For example, some metrics can be computed during the analysis phase as the number of ambiguities or inconsistencies discovered pertaining to security requirements. Some can be computed at the design phase. Generally, metrics that can be computed at the earlier phases are considered to have more utility as if the computed values of such metrics indicate anything wrong, corrections can be applied immediately and it is generally easy to make corrections at early stages.

An exhaustive attempt to develop security metrics pertaining to all the phases of software engineering is beyond the scope of this research. Instead this research will attempt to propose metrics at the source code level. Source Code is one of the most attractive candidates for security measurement as it is the direct embodiment of the software. Many metrics that attempt to measure the quality of the software using the source code have been proposed and studied (Halstead, 1977; Mills, 1988). Many static analysis tools that automatically scan the source code and compute the metrics have also been developed (“FindBugs”, n.d.). These metrics have the only
disadvantage that they can be computed only after the programs have been developed which is a late part in the software engineering process. In attempts to develop security metrics, those relating to the source code seem to be the most easily obtainable and objective. In this context, Voas et al propose a fault injection based security metric (Voas et. Al., 1996) and the research proposes an enhancement to this method by using Mutation Testing. The research also demonstrates empirically that such an enhancement improves the accuracy of the measured security.

The research also proposes some metrics that can be computed for methods written in Java. Various static characteristics of the method are taken into account before arriving at a quantitative indicator of the level of security (or vulnerability) for the method. This stands in contrast to the previous attempt where the dynamic or run-time characteristics of programs are studied to compute a security indicator. The proposed metrics are validated by applying them to real time java applications and drawing correlations between the computed values for the metrics and the number of attacks on the application that exploit vulnerabilities in the method under consideration.

Finally, the research applies a Genetic Algorithm to identify the most effective subset of Software Security Metrics that can throw light on the Security of a software application based on actual data available pertaining to attacks on a Web Site that has been in operation for over a year.

1.8 Organization of the Thesis

The thesis is organized as follows:

Chapter 2 reviews the related literature in the domain of Security Metrics. It briefly describes various researches on Security Metrics and throws light on their salient features.
Chapter 3 explores the possibility of using many existing, well-documented metrics for measuring Security.

Chapter 4 proposes a fault-injection based Security Metric that is based on an enhancement of a technique called Adaptive Vulnerability Analysis. The value addition provided by the enhancement is validated by an empirical study.

Chapter 5 proposes several metrics that can be computed at the level of methods for Java Programs. The metrics are applied to real time programs and the results shown.

Chapter 6 utilizes a Genetic Algorithm to identify the subset of metrics that are indicative of the Security of a Software.

Chapter 7 concludes the research and provides directions for future enhancements to the research.