Security Metrics for Source Code Structures

Istehad Chowdhury
Department of Electrical and Computer Engineering
Queen's University, Kingston
Ontario, Canada, K7L3N6
istehad@cs.queensu.ca

Brian Chan
Department of Electrical and Computer Engineering
Queen's University, Kingston
Ontario, Canada, K7L3N6
2byc@queensu.ca

Mohammad Zulkernine
School of Computing
Queen's University, Kingston
Ontario, Canada, K7L3N6
mzulker@cs.queensu.ca

ABSTRACT

Software security metrics are measurements to assess security related imperfections (or perfections) introduced during software development. A number of security metrics have been proposed. However, all the perspectives of a software system have not been provided specific attention. While most security metrics evaluate software from a system-level perspective, it can also be useful to analyze defects at a lower level, i.e., at the source code level. To address this issue, we propose some code-level security metrics which can be used to suggest the level of security of a code segment. We provide guidelines about where and how these metrics can be used to improve source code structures. We have also conducted two case studies to demonstrate the applicability of the proposed metrics.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics – *Performance measures and Product metrics.*

General Terms

Measurements and Security.

Keywords

Metrics, security metrics, code quality and security.

1. INTRODUCTION

By any measure, security holes in software are common and the problem is growing. Much of the previous research related to security and software has focused on the protection mechanisms, i.e., application security, and the importance of software security is recently gaining awareness [1]. "Application security is about protecting software and the systems that software runs in a post facto way, after development is complete" [5]. Software security is about building secure software, taking security concerns from the very beginning of software development life cycle [5]. We define software security metrics as measurements to assess security related imperfections (or perfections) introduced during system development. The metric-based quantitative methods can permit resource allocation for achieving a desired security level. The saying of Lord Kelvin -"if you cannot measure it, you do not know it"- captures the whole essence of the metric-based approach of security estimation.

If there is no malicious intent, there will be no need for security solutions. The malicious intent can be so strong that even slightly imperfect source code can be exploited to cause security breaches. Therefore, the potential of damage by an attacker to exploit the code quality may require focused attention. In this work, we have tried to focus on this issue. Instead of only looking from a higher level of abstraction (i.e., from system level perspective), we try to measure source code quality that may lead to enhancing program security – "security at program level" as defined by Pfleeger [9].

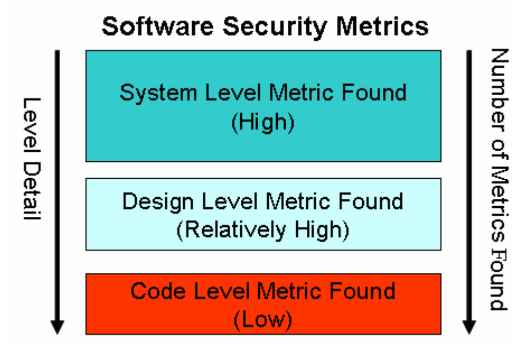

Software Security Metrics
Level Detail
System Level Metric Found (High)
Design Level Metric Found (Relatively High)
Code Level Metric Found (Low)
Number of Metrics Found

Figure 1: Current state of metric types found

A survey has revealed that most of the security metrics assess the security status at system level [2], [3], [6], [7]. System-level security metrics attempt to quantify the security status of a system as a whole. They are referred to as high-level metrics in the literature [6]. These metrics look into not only software but also many other aspects of the system. They consider issues from current system configuration to even percentage of employees with significant security responsibilities who have received specialized training. For example, system vulnerability index (SVI) is calculated by evaluating factors such as "system characteristics", "potentially neglectful acts" and "potentially malevolent acts"— as a measure of computer system vulnerability [4]. On the other hand, design level security metrics assese the security issues pertaining to software artifacts at design phase. Code-level security metrics specifically look into code structures and implementation language issues. As we go down to lower level of abstraction, there seems to be fewer and fewer metrics. Figure 1 highlights this existing scenario.

To address this gap, we propose some metrics that can assess how securely a system's source code is structured. The metrics are stall ratio, coupling corruption propagation, and critical element ratio. We conduct two case studies to demonstrate how the proposed metrics can be applied to evaluate the robustness, secure information flow, and secure control flow in their code structures. One of the software under testing is acknowledged to be fairly well-coded and more secure than the other one. The proposed metrics also reveal the contrast between the two pieces of software under testing concerning their secure code structures. Therefore, the case studies demonstrate that the proposed metrics reflect some intended aspect of security of software systems.

The rest of the paper is organized as follows. Section 2 describes some of the related work. Section 3 provides a detailed description of the proposed metrics. Section 4 presents the results of the conducted case studies and discusses the implications of the obtained metric values. Finally, Section 6 concludes the paper and outlines some avenues for future work.

2. RELATED WORK

Literature review reveals that two measurements are commonly used to determine the security of a system: (1) counting the number of security related bugs found (or fixed from one version to the next), and (2) counting the number of times a system's version mentioned in CERT advisories [12], Microsoft Security Bulletins [13], MITRE Common Vulnerabilities and Exposures (CVE) [14], Bugzilla etc. Both approaches are quite useful, while they have their drawbacks. The problem with the first approach is that it may miss some bugs, and equal importance is given to all the bugs [7]. The latter approach ignores the specific system configuration that causes vulnerabilities [6]. It also does not include the possibility of successful future attacks on the system [15].

Based on the number of times a software system is mentioned in different security bulletins, Alhazmi *et al.* [2] propose a new metric called vulnerability density, (V_D). V_D is defined as the number of vulnerabilities in the unit size of code. Using this as the parent metric, a suite of security metrics are coined such as known vulnerability density (V_{KD}), vulnerability discovery rate, residual vulnerability density (V_{RD}), and ratio of vulnerable density to defect density (V_D/D_D). V_{KD} is the number of the already identified vulnerabilities. Vulnerability discovery rate is the number of vulnerabilities reported/identified per unit time. Residual vulnerability density is defined as, $V_{RD} = V_D - V_{KD}$.

There have been some more attempts in quantifying the security aspect of software systems. Brocklehurst *et al.* [3] measures the operational security of a system by estimating the effort spent by an attacker to cause a security breach in the system and the reward associated with the breach. As already mentioned, system vulnerability index (SVI) is calculated by evaluating factors such as "system characteristics", "potentially neglectful acts", and "potentially malevolent acts" [4]. SVI can be computed by adding, multiplying the factors or normalizing the results in some way. However, the authors do not provide specific guidelines about how to retrieve those factors from a given system and assign values to the factors that comprise SVI. All of the aforementioned metrics quite successfully depict the broader security picture of a system at a higher level of abstraction than the code-level.

Minimum-time-to-intrusion (MTTI) metric is defined as "the predicted period of time before any simulated intrusion can take place" [7]. MTTI is a relative metric that allows the users to compare different versions of the same system. However, MTTI focuses on the operational security aspect, i.e., on "application security".

Manadatha *et al.* [6] uses an intuitive approach to measure a system's attack surface, the set of ways in which an adversary can attack the system. In other words, attack surfaces are the areas of the software more susceptible to attacks. The "attack surface metric strikes at the design-level of a system: above the level of code, but below the level of the entire system". Our proposed critical element ratio metric is analogous to this attack surface metric. The attack surface metric measures the set of ways in which an adversary can attack the system; whereas the critical element ratio metric measures the ratio of elements that are corruptible by malicious user inputs to the total elements in a class or method. Thus, the proposed critical element ratio metric works at a much lower level than that of attack surface metric.

The coupling corruption metric is inspired by an object oriented design level metric called response set of a class (RFC), defined as the set of all methods that can be invoked in response to a message to an object of the class [10]. We take the idea of RFC and apply a suitable security concern. While the concept of RFC is not new, the proposed security components are unique.

The closest example of a code-level metric is associated with exception handling. Aggarwal *et al.* [8] have coined this idea of "Exception Handling Ratio" and defined it as the ratio of the number of catch statements declared to the number of catch statements possible in a class. The authors imply that a low exception handling ratio indicates a poor design quality, and eventually, this vulnerability can be exploited by an attack.

3. PROPOSED METRICS

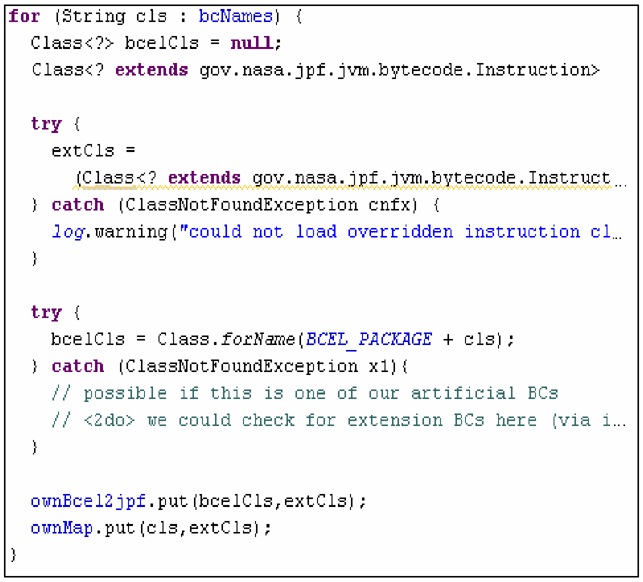
How can we assess the security of a code segment? The proposed metrics address this question in an indirect way. The metrics measure source code quality properties that can enhance program security. In effect, the metrics are designed to measure the soundness of source code structures. Any metric will be rendered useless if an attacker has full control of the software. The primary objective of the proposed measurement approaches is to quantify whether an attacker can produce any undesired effect in the program by exploiting the imperfections in the source code structures.

3.1 Stall Ratio (SR)

Stall ratio is a measure of how much a program's progress is impeded by frivolous activities. In a program, certain statements may not be strictly necessary to make a program progress towards its desired goal. These statements may cause delay to a program to reach an accepting state (e.g., allowing a counter value to reach 10). In a loop, there may be statements that do not contribute to the overall progress of the program such as performing frivolous calculations (e.g., a=a+0). We term this type of frivolous activities in a program as stall statements. The delay caused by the stall statements becomes more significant especially when they are in a loop. Therefore, the proposed metric to measure the progress of a program is defined as follows:

$$\text{Stall ratio} = \frac{\text{Lines of nonprogressive statements in a loop}}{\text{Total lines in the loop}}$$

The code snippet of Figure 2 illustrates how stall ratio of a loop is computed. By the definition of stall ratio, any statement in a loop that does not help bring the program to the final state is considered a stall statement. In this case, the *log.warning* is considered such a statement as it is not strictly necessary to log error messages until an error has been detected. However, all the other statements are essential for the program to reach the goal state. There are a total of 12 executable statements (excluding the comments), and only one of them is identified as a stall statement. Therefore, the stall ratio for this piece of code is 0.083 (1/12).



```
for (String cls : bcNames) {
  Class<?> bcelCls = null;
  Class<? extends gov.nasa.jpf.jvm.bytecode.Instruction>

  try {
    extCls =
      (Class<? extends gov.nasa.jpf.jvm.bytecode.Instruct…
  } catch (ClassNotFoundException cnfx) {
    log.warning("could not load overridden instruction cl…
  }

  try {
    bcelCls = Class.forName(BCEL_PACKAGE + cls);
  } catch (ClassNotFoundException x1){
    // possible if this is one of our artificial BCs
    // <2do> we could check for extension BCs here (via i…
  }

  ownBcel2jpf.put(bcelCls,extCls);
  ownMap.put(cls,extCls);
}
```

Figure 2: Code snippet to illustrate the computation of SR

It is evident that stall ratio is a measure of tardiness of a code segment. All stall statements are not necessarily "bad". However, one should be aware which segments of the code have proportionately more stall statements. It is necessary to write logs, report errors messages and print messages for the operation of a software system. Nevertheless, this should not be done so often that these apparently not so important activities become a threat to the overall performance of the system by delaying the response of the program. One common method of attack involves saturating the target (victim) machine with external communication requests. In that situation, the victim cannot respond to legitimate traffic or it responds so slowly that can be rendered effectively unavailable [1]. Similar attack can be attempted to exploit code sections with high stall ratio. Let us suppose, continuous attacks come from multiple computers causing an excessive execution of this log-writing section of the code. Then, the software will be so busy in writing logs (i.e., doing I/O operations) that the overall performance may degrade. Stall ratio may initially seem to be insignificant, but it is indeed an important issue in performance, security, and mission critical environments.

3.2 Coupling Corruption Propagation (CCP)

Coupling between methods is the concept that two or more methods are reliant on each other in some way. This could involve data sharing or decision making in the child methods using one of its call parameters. The effects of coupling can easily ripple into other methods several levels down the call chain. Coupling corruption propagation is meant to measure the total number of methods that could be affected by erroneous originating method. An arithmetic error can result when certain ranges of parameter values can be accepted in one method, but this range of values may exceed limits in another method that takes the same parameter. Such a situation can result in exception errors as well. Another potential security flaw results when a critical parameter is forced to remain at a certain value, and the result remains the same no matter what other parameters are changed. This can result in restriction of services. The formal definition of this proposed metric is:

Coupling Corruption Propagation = Number of child methods invoked with the parameter(s) based on the parameter(s) of the original invocation

```
Public void CalculateStringBuildSpeed(double
numOfElements)
{
  buildString (numOfElements);
}
.
.
.
public String buildString(double length) {
  String result = "";
  int startTime;
  int endTime;
  int stringLength = (int) length;

  startTime = callStartTime();
  System.out.println("Starting Now");
  for (int i = 0; i < stringLength; i++) {
    result += (char)(i%26 + 'a');
  }
  endTime = callEndTime();
  reportResult(startTime,endTime,stringLength);
  return result;
}
```

Figure 3: Code snippet to illustrate the computation of CCP

The code snippet of Figure 3 illustrates how CCP is calculated. In Figure 3, a parent method *calculateStringBuildSpeed* invokes a child method called *buildString* with the parameter *length*. The method *buildString* then calls three child methods of its own *callStartTime*, *callEndTime,* and *reportResult*. The methods *callStartTim* and *callEndTime* do not take any parameter. Therefore, they are effectively immune to any coupling effects that originate from *calculateStringBuildSpeed*. However, *reportResult* takes one variable *stringLength* that is calculated from the original *length* variable. If the variable is ever compromised (i.e., it becomes null, has a value that exceeds the *int* range, or contains a value that *reportResult* cannot use), then *reportResult* method could indirectly be corrupted from *calculateStringBuildSpeed*. Therefore, the coupling corruption propagation is 2.

Coupling is often inevitable. However, highly coupled classes require vigorous testing as there is a possibility of deeper damage propagation [10]. With a lower coupling, the CCP is also low by definition. Therefore, with CCP, we can estimate the extent of the damage propagation when some data elements are compromised by a successful attack.

3.3 Critical Element Ratio (CER)

Object oriented code contains certain data objects that must be instantiated during runtime. However, a process may not require all aspects of an object to be instantiated (i.e., they can remain NULL for the whole duration of the program runtime). A security risk emerges if some essential data objects are changed that may destabilize the process as a whole. Hence, the more critical elements in a class, the higher is the security risk. This risk is quantified in the following way:

$$\text{Critical Elements Ratio} = \frac{\text{Critical Data Elements in an Object}}{\text{Total Number of Elements in the Object}}$$

```
public class MethodTester{
  static   final   ArgList   EMPTY_ARGS   =   new
  ArgList();

  String[] mthPatterns;
  StringSetMatcher mthMatcher;

  PrintWriter err;
  PrintWriter log;
  TestContext tctx;

  Class<?> targetCls;
  Constructor[] targetCtors;

  int TotalMth, nTestMth;
  int nTests;
  int nOk, nFailure, nMalformed;

  void showStatistics(){
    log("statistics:");
    if(nTestMth>0)
    {
        log("tested methods: " + nTestMth + "
        of " + nTotalMth);
        log("test: " + nTests);
        log("passed: " + nOk);
    }
    if(nMalformed>0){
        log("MALFORMED:    "+ nMalformed);
    }
    if(nFailure >0){
        log("FAILED:       "+ nFailure);
    }
  }
}
```

Figure 4: Code snippet to illustrate the computation of CER

Here, a critical data element is the one that must eventually be used through the course of the program. Let us demonstrate the determination of critical element ratio with an example. In Figure 4, there are 13 elements of the *MethodTester* class. All the variables which are not *int* type are used in critical components in the program. They are either assigned values or are used in complexity components (i.e., decisions/method passing).

In the function *ShowStatistics,* we observe that the *int* type variables are used for internal logging purpose and are not strictly critical for the completion of the main objective of the program. Even the default values of the *int* variables can be logged and the instantiation of the variables are not strictly required for the "safe" continuation of the program. That is, if these values are compromised (given different values), they will not jeopardize the entire program. Therefore, there are 6 non-critical elements (out of a total of 13 elements). Hence, the critical element ratio is 0.5385 (7/13).

Some data elements are user provided and some are generated by the system. The critical element ratio metric measures the ratio of elements that can be possibly corrupted by malicious user inputs to the total elements in a class or method. The more such user inputs enter the system, the more the system is open to the user. The more the system is open to the users, the more is the risk of getting attacked by malicious user inputs (given that all other parameters remain constant). Hence, the classes or components with higher critical elements should be tested more carefully than that with lower critical elements. Moreover, components with higher critical elements require stronger input validation schemes. Given that, a manager can use this CER metric to better manage his or her test plans and allocate resources.

4. CASE STUDIES

We demonstrate the effectiveness and applicability of the proposed metrics in this section. Java Pathfinder (JPF) and JDemo Launch are the two pieces of Eclipse plug-ins chosen for our case studies. Eclipse plug-ins are chosen for their relative compactness while they can also provide varied functionality. They do not require an extensive environment setup to be executed. The plug-ins also allow the user to easily view the debugger information showing the system runtime values. A very brief overview of the selected plug-ins is given as follows:

1. JavaPathFinder (JPF): The Java PathFinder, JPF, is a translator from Java to Promela, the programming language of the SPIN model checker. The purpose is to establish a framework for verification and debugging of Java programs using model checking. The system detects deadlocks and violations of assertions stated by the programmer.

2. JDemo: This plug-in provides a new approach of demo driven development: when developing software, we write short code snippets (demo cases) that use our new API. JDemo then demonstrates both: how to use the API and what happens when we execute the code. In this sense, it acts as a debugger. We can also interactively test the usability of GUI components using JDemo.

JPF is chosen for its relatively professional coding style. This tool is developed and utilized in NASA to verify their programs for deadlocks. A large concurrent-version-system (CVS) bug repository and message board for questions regarding the JPF are available. This gives an impression that Java Pathfinder should be fairly well-coded and secure. On the other hand, JDemo is rather low-profile software developed by a small number of freelance developers. It is hoped that a comparison between the two pieces of software would reveal their relative security levels when analyzing them using the proposed metrics.

4.1 Selection of an Existing Metric

We use an existing code-level security metric to show the correlation among the existing and the proposed metrics. We have observed a similar approach of “cross-checking” in [11]. For our case, the selected existing metric is the number of catch blocks per class (NCBC) [8]. The NCBC metric measures the exception handing factor (EHF) in some way, and NCBC and EHF are used synonymously in this paper. It is defined as the percentage of catch blocks in a class over the theoretical maximum number of catch blocks in the class. However, we present the EHF as a ratio rather than as a percentage. We do so to keep it comparable to some of the proposed metrics which are also expressed as ratios. The following equation is the formal definition of this metric.

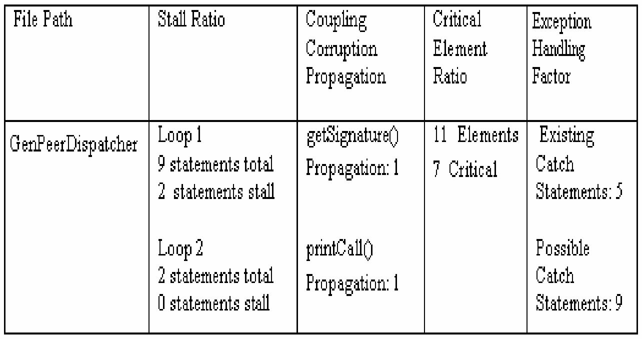
$$\text{NCBC (or EHF)} = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m} C_{ij}}{\sum_{i=1}^{n}\sum_{k=1}^{l} C_{ik}} \times 100$$

The variable n is defined as the number of methods in a class, and m is the number of existing catch statements in that particular method n. The variable l is the maximum number of possible catch statements that could be included within that class. Therefore, the numerator represents the total number of existing catch statements in the class, and the denominator indicates the total number of possible catch statements. In this case, $C_{ij} \le C_{ik}$.

4.2 Data Collection Methodology

We have chosen the class files that are evenly distributed across the components for each software. This is done to ensure that the mean metric values are not skewed to one particular component in the application. In each class file, the associated code structures are analyzed to compute the metrics values. A sample of the detail data gathered to compute the SR, CCP, CER, and EHF for a class is shown in Table 1.

Table 1: A sample of data collected per class file



File Path	Stall Ratio	Coupling Corruption Propagation	Critical Element Ratio	Exception Handling Factor

GenPeerDispatcher	Loop 1
	9 statements total
	2 statements stall
	
	Loop 2
	2 statements total
	0 statements stall
		getSignature()
		Propagation: 1
		
		printCall()
		Propagation: 1
			11 Elements
			7 Critical
				Existing Catch Statements: 5
				
				Possible Catch Statements: 9

To find the stall ratio, loops in the class file are identified and their total size is counted. The loops are then analyzed for potential stall statements that can deter the overall progress.

Coupling corruption involves analyzing all function calls originating in every class file. If the parent function call has parameters that are subsequently used in decisions in child methods, then the count of total propagation for that parent method is increased. This process is repeated for each method until no more control flows are found. For example in Table 1, the file *GreenPeerDispatcher* has two methods *GetSignature()* and *PrintCall()*. Both the methods have coupling corruption propagation equal to 1. Hence, the CCP for the overall file is 2.

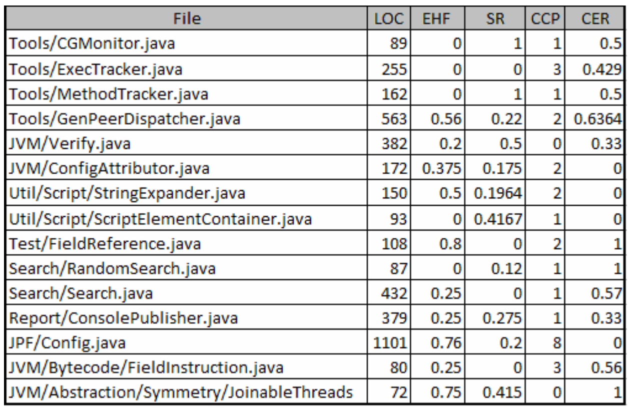
The critical element ratio is calculated by observing the internal variables for each class. The total number of elements for a class is recorded and then each element is analyzed to see whether they are used to calculate subsequent values. Elements that have default values and are used to print messages or store error messages are considered as non critical.

The exception handling factor for each class is determined based on the definition provided in [8]. Possible catch statements can be defined for every situation. It is up to the user’s discretion and situation to define which catch statements could have been included in a particular class. For the majority of the files, these catch statements are of three types: null-pointer exception, array-out-of-bounds exception, and illegal-arithmetic-operation exception. This process is applied to all chosen files for each case study. By using this generic pattern for analysis, the goal is to get the overall security quality for the system itself.

4.3 Analysis of JavaPathFinder Data

The metric values for the JPF are analyzed to assess the secure code structures of the software. Table 2 presents the LOC, EHF, SR, CCP, and CER metric for each file of the JPF. Since JPF is fairly larger than JDemo, the lines of code (LOC) metric is recorded to observe whether the security metrics values are dependent on the file sizes. The mean and median of each of the metrics are summarized in Table 3. If the mean is significantly higher than the median, this would mean that there is abrupt presence of high outliers and vice versa. For example, the mean CCP is almost double the median CCP for JPF. In most cases, however, the mean and median are relatively close.

Table 2: Metric values of JPF



File	LOC	EHF	SR	CCP	CER
Tools/CGMonitor.java	89	0	1	1	0.5
Tools/ExecTracker.java	255	0	0	3	0.429
Tools/MethodTracker.java	162	0	1	1	0.5
Tools/GenPeerDispatcher.java	563	0.56	0.22	2	0.6364
JVM/Verify.java	382	0.2	0.5	0	0.33
JVM/ConfigAttributor.java	172	0.375	0.175	2	0
Util/Script/StringExpander.java	150	0.5	0.1964	2	0
Util/Script/ScriptElementContainer.java	93	0	0.4167	1	0
Test/FieldReference.java	108	0.8	0	2	1
Search/RandomSearch.java	87	0	0.12	1	1
Search/Search.java	432	0.25	0	1	0.57
Report/ConsolePublisher.java	379	0.25	0.275	1	0.33
JPF/Config.java	1101	0.76	0.2	8	0
JVM/Bytecode/FieldInstruction.java	80	0.25	0	3	0.56
JVM/Abstraction/Symmetry/JoinableThreads	72	0.75	0.415	0	1

In general, the coupling propagation for the Java Pathfinder system is relatively low. One method is calling (with parameter) fewer than two other methods on average. This reflects a secure coding style because a lower CCP restricts the possibility of damage propagation to many other components. A closer look at Table 2 reveals that some of the larger class files have a higher coupling propagation. This observation is also supported by the positive correlation between LOC and CCP reported in Table 4. A unique trend in JPF is the relatively high stall ratio. Almost 30%

of the statements in JPF are stall statements (mean SR is 0.3). JPF is designed to report errors to the user interface as well as log errors for future analysis. Based on the stall ratio definition, this is treated as unnecessarily time consuming as it delays the program from reaching the end state. This can be an explanation of the high stall ratio in JPF. Almost 46% (mean CCP is 0.46) of the elements in JPF are critical. This is fairly significant. The non-critical elements are usually variables that have default values and are used in print statements.

Table 3: Mean and median of metrics of JPF

	LOC	EHF	SR	CCP	CER
Mean	275	0.31	0.30	1.87	0.46
Median	162	0.25	0.20	1	0.50

Table 4: Correlation among metrics of JPF

	LOC	EHF	SR	CCP	CER
LOC	1	0.38	-0.17	0.72	-0.35
EHF	-	1	-0.36	0.38	0.13
SR	-	-	1	-0.32	-0.08
CCP	-	-	-	1	-0.38
CER	-	-	-	-	1

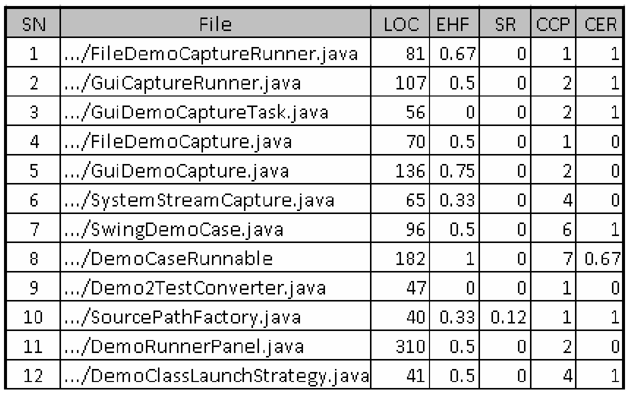
Finally, the correlations among the metrics are presented in Table 4. Correlation among the metrics may potentially reveal any interesting pattern about the code structure of the software and whether there is a noticeable interdependence between any given pair of metrics. The EHF is negatively correlated to the stall ratio metric. However, it is positively correlated to the coupling corruption propagation and critical element ratio. There is also a strong positive correlation among LOC, EHF, and CCP.

4.4 Analysis of JDemo Launch Data

We analyze the metric data of JDemo and compare with that of the JPF. Table 5 presents the LOC, EHF, SR, CCP, and CER metric for each file of JDemo, and Table 6 reports the mean and median for each of the metrics. On average, the lines of code in JDemo are considerably fewer than that in Java PathFinder. This observation coincides with the assumption that the former system is overall less complex.

The EHF of JDemo is also marginally higher even though the average file size is lower than that of JPF. However, Table 7 indicates that there is a strong correlation between LOC and EHF. This indicates that exceptions are well-handled in larger class files. The CCP in JDemo is significantly higher than that in JPF. It implies that the top level messages in JDemo have a higher chance of propagating to other components in the system. This higher CCP can cause a potentially larger security breach in Jdemo. Moreover, the CER in JDemo is also significantly higher than that in JPF. This is undesirable because there is a higher chance of a critical element being modified, and it leads to a greater possibility of the system reaching an undesirable state. The stall ratio for JDemo is very low compared to JPF. This can be explained by JDemo not having to report most of the errors as JPF has to do. Therefore, JDemot does not stall that much.

Table 5: Metric values of JDemo



SN	File	LOC	EHF	SR	CCP	CER
1	.../FileDemoCaptureRunner.java	81	0.67	0	1	1
2	.../GuiCaptureRunner.java	107	0.5	0	2	1
3	.../GuiDemoCaptureTask.java	56	0	0	2	1
4	.../FileDemoCapture.java	70	0.5	0	1	0
5	.../GuiDemoCapture.java	136	0.75	0	2	0
6	.../SystemStreamCapture.java	65	0.33	0	4	0
7	.../SwingDemoCase.java	96	0.5	0	6	1
8	.../DemoCaseRunnable	182	1	0	7	0.67
9	.../Demo2TestConverter.java	47	0	0	1	0
10	.../SourcePathFactory.java	40	0.33	0.12	1	1
11	.../DemoRunnerPanel.java	310	0.5	0	2	0
12	.../DemoClassLaunchStrategy.java	41	0.5	0	4	1

Table 6: Mean and median of metrics of JPF

	LOC	EHF	SR	CCP	CER
Mean	103	0.47	0.01	2.75	0.56
Median	75.5	0.50	0	2	0.83

Table 7: Correlation among metrics of JDemo

	LOC	EHF	SR	CCP	CER
LOC	1	0.46	-0.25	0.20	-0.33
EHF	-	1	-0.15	0.47	0.04
SR	-	-	1	-0.27	0.28
CCP	-	-	-	1	0.21
CER	-	-	-	-	1

The correlation between EHF and the proposed metrics are similar to the results with JavaPathfinder. We observe this from the correlations presented in Table 7. In both the case studies, LOC is positively correlated with EHF and CCP, and negatively correlated with CER. EHF and CCP are also consistently positively correlated.

4.5 Overall Discussion

The data analysis of the two products shows that JavaPathfinder is overall more secure than JDemo. This confirms with the initial assumption that JPF would be more secure because it is utilized in a more professional environment, and it is subjected to more intense scrutiny in terms of how it should perform.

The JPF, having a lower CCP, it is less likely to bring top level messages to a lower depth. This is an indication of safe information and control flow structure. The average file size in JPF is relatively larger but it still refrains from allowing control values propagating too far.

The correlation between EHF and SR is negative for both JPF and JDemo Launch. This can be viewed as a security pattern of how the code should be structured. That is, developers that structure their code better are more concerned about security and they are less likely to perform frivolous activities. However, this correlation is not strong enough to warrant this claim.

For both the software, the correlation patterns between lines of code and the other metrics are similar. LOC shows a strong positive correlation to EHF and also a significant negative correlation to CER. It can be inferred from the result that the difference in file size does not have significant effect on the comparability of the metric values obtained from the two pieces of software.

The proposed metrics have a strong correlation with the exception handling factor metric (EHF). In this case, both SR and CCP are significantly correlated to EHF. It should be noted that strong correlation between two metrics does not necessarily mean that the metrics measure the same aspect of the program. Overall, we have observed some repetitive correlation patterns in both case studies. For example, in both cases, SR is negatively correlated to EHF and positively correlated to CER. This may indicate that certain rules should hold consistently (i.e., a system with more EHF should employ fewer critical elements or higher CCP means more catch statements need to be implemented to catch undesirable outcomes). This consistent pattern promises that these metrics can be employed to analyze code structures across systems.

The proposed metrics can help to bring improvement in code structures and also help to manage test distribution. Classes with higher CER and/or higher CCP need to be tested more vigorously as they are more complex and more prone to attacks from malicious user inputs. Because CCP is strongly correlated to EHF, CCP can indicate the required degree of exception handling mechanism. Let us suppose that Version 2 of a particular piece of software has higher CCP and higher CER than that of Version 1. One would generally expect an improvement in the exception handling factor of Version 2 as well. Otherwise, Version 2 may be more susceptible to attacks and prone to failures. We know that something should be done about stall ratio if there is a need to increase the response time of the system, and the system can be monitored to observe if the taken measures improve the response time.

5. CONCLUSION AND FUTURE WORK

As part of this research, we have conducted an exhaustive survey of the commonly reported security metrics in the literature. To address the issue of relatively fewer metrics observed in the source code level, we have proposed three metrics: stall ratio, coupling corruption propagation, and critical element ratio. The stall ratio finds the ratio of stall statements in loop structures which can be exploited to cause a denial of service attack at the worst case. Coupling corruption propagation measures how far a potential damage may propagate to the other components of the software. The extent of damage caused by an attack can be more serious for software with high CCP. The critical element ratio measures how many ways a program or class can be infected by malicious inputs. The higher the critical element ratio in a system, the more the system is open to attacks.

We have conducted two case studies to demonstrate how the proposed metrics can be effectively used for different measurements. We make some interesting observations as well. The correlations among the metrics demonstrate some consistent patterns. For example, in both the software under study, the exception handling factor is higher in the classes with relatively higher coupling corruption propagation. The consistent patterns suggest that these phenomena are not observed by chance.

One of the major challenges in proposing a new metric is devising a comprehensive evaluation scheme. We have tried to evaluate our metrics by performing some case studies. We observe that the results of the case studies conform to the agreed upon assumption that JPF has more secure coding structure than JDemo. Another issue is that some of the metric values are decided based on intuition. For example, finding out the critical elements in a class depends on the intuition of the data collectors. However, this approach is not new and has been observed in previous research in security metrics. For example, data for attack surface metric has also been collected based on this type of heuristic [6].

The future plan is to automate the computation of the metrics as far as possible so that it does not become tedious for large software. We also plan to conduct more experiments to draw more concrete conclusions about the implications of the metric values. While each metric has its own strengths and weaknesses, the objective of this work is to come up with more code-level metrics as we have observed that relatively fewer metrics assess security at code level.

6. ACKNOWLEDGEMENT

This research is partially funded by the Natural Sciences and Engineering Research Council of Canada (NSERC).

7. REFERENCES

[1] Adams, C. and Jourdan, G.V. 2005. Why Good Software Engineering Practices Often Do Not Produce Secure Software. Workshop on Cyber Infrastructure – Emergence Preparedness Aspects (Ottawa, Ontario, Canada, Apr. 2005).

[2] Alhazmi, O.H., Malaiya, Y.K., and Ray, I. 2007. Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers and Security Journal* 26, 3 (May 2007), 219-228.

[3] Littlewood, B., Brocklehurst, S., Fenton, N., Mellor, P., Page, S., and Wright, D. 1993. Towards operational measures of computer security. *Journal of Computer Security* 2, 3 (1993), 211–230.

[4] Alves-Foss, J. and Barbosa, S. 1995. Assessing computer security vulnerability. *SIGOPS Oper. Syst. Rev.* 29, 3 (Jul. 1995), 3-13.

[5] McGraw, G. 2006. *Software Security: Building Security In*. HHAddison-WesleyHH.

[6] Manadhata, P. K. and Wing, J. M. 2005. An Attack Surface Metric. Technical Report. School of Computer Science, Carnegie Mellon University (CMU). CMU-CS-05-155.

[7] Voas, J., Ghosh, A., McGraw, G., Charron, F., and Miller, K. 1996. Defining an Adaptive Software: Security Metric from a Dynamic Software Failure Tolerance Measure. In *Proceedings of the Annual Conference on Computer Assurance* (Gaithersburg, MD, USA, June 1996). 250-263.

[8] Aggarwal, K.K., Singh, Y., Kaur, A., and Malhotra, R. 2006. Software Design Metrics for Object-Oriented Software. *Journal of Object Technology* 6, 1 (Jan. 2006), 121-138.

[9] Pfleeger, C. and Pfleeger, S. 2003. *Security in Computing*. Prentice-Hall Inc.

[10] Chidamber, S. R. and Kemerer, C. F. 1994. A Metrics Suite for Object Oriented Design. *IEEE Trans. Softw. Eng.* 20, 6 (Jun. 1994), 476-493.

[11] Khaer, M. A., Hashem, M. M. A., and Masud, M. R. 2007. An Empirical Analysis of Software Systems for Measurement of Design Quality Level Based on Design Patterns. In *Proceedings of the 10th International Conference on Computer and Information Technology*. (Dhaka, Bangladesh, Dec. 2007), In Press.

[12] Carnegie Mellon University's Computer Emergency Response Team (CERT) Advisories, http://www.cert.org/advisories.

[13] Microsoft Security Bulletins, http://www.microsoft.com/technet/security/current.asp

[14] MITRE Common Vulnerabilities and Exposures (CVE), http://www.cve.mitre.org.

[15] Howard, M. 2003. Fending Off Future Attacks by Reducing Attack Surface, Technical Report, HHhttp://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure02132003.aspHH.