VULNERABILITY ASSESSMENT OF JAVA BYTECODE

Except where reference is made to the work of others, the work described in this thesis is my own or was done in collaboration with my advisory committee. This thesis does not include proprietary or classified information.

_____

Rahul Arvind Shah

Certificate of Approval:

_____	_____
Drew Hamilton Associate Professor Computer Science and Software Engineering	David A. Umphress, Chair Associate Professor Computer Science and Software Engineering
_____	_____
Dean Hendrix Associate Professor Computer Science and Software Engineering	Stephen L. McFarland Acting Dean Graduate School

VULNERABILITY ASSESSMENT OF JAVA BYTECODE

Rahul Arvind Shah

A Thesis

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements for the

Degree of

Master of Science

Auburn, Alabama

December 16, 2005

VULNERABILITY ASSESSMENT OF JAVA BYTECODE

Rahul Arvind Shah

_____
Signature of Author

_____
Date of Graduation

THESIS ABSTRACT

VULNERABILITY ASSESSMENT OF JAVA BYTECODE

Rahul Arvind Shah

Master of Science, December 16, 2005
(B.Sc. Physics, Mumbai University, 1999)

182 Typed Pages

Directed by David A. Umphress

Security of the software applications has become a critical issue as software is now used in almost all sectors parts of our day to day life. There is always an underlying threat that a malicious user may be able to access classified information, intellectual information or secret algorithms by exploiting the software applications in many possible ways. The research described here examines the possible security threats to any stand-alone software applications developed in Java. The Java bytecode adheres to a well-defined class file format as described in the JVM specifications, and this makes the bytecode more vulnerable. The bytecode vulnerability taxonomy is developed and can be used to increase our overall understanding of the bytecode vulnerabilities. The focus of this research is to conduct a vulnerability assessment of Java bytecode in order to reveal its vulnerabilities. As part of case study, the class files are exploited to carry out intellectual penetration and component penetration attacks followed by the validations.

ACKNOWLEDGEMENTS

I would like to thank Dr. David Umphress, my major professor, for his support and encouragement throughout this research. When things were at their most difficult, he challenged me and helped me to achieve more than I thought possible. I am also thankful to Dr. Drew Hamilton and Dr. Dean Hendrix for being in my committee and giving time and comments to the work done.

Finally, I would like to thank my family, who has provided the stability and strength that have allowed me to complete this thesis. I could not have done without you.

Style manual or journal used: ACM Digital Library

Computer software used: Microsoft Office XP

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION AND PROBLEM STATEMENT

Security of software applications has become a critical issue as software is now used in almost all sectors of our community and in the business world. As a result, it is no longer possible to ignore the serious security concerns associated with many software applications. Security of information technology is a broad domain involving many different types of information security, such as computers and system security, network security, software application security, and data security. Research in the field of information security has grown tremendously in recent years and there is now an enormous amount of literature describing the tools and methods that have been developed to cope with the threats to information security. Most of this research has concentrated on network and computer security, with a continuous stream of new developments in Internet technology, mobile and pervasive computing. However, most of the effort devoted to assessing security risks of software applications has been limited to the vendors and industries that use particular software platforms.

It is our general tendency to protect sensitive data, packets floating around networks, and other computer resources, but we cannot underestimate the threat on any stand-alone software applications because these software programs work along with other technologies, and wrap and manipulate the sensitive data. There is always an underlying threat that a malicious user may be able to access classified information, intellectual information or secret algorithms by exploiting the software applications in many possible

ways. Software applications can thus become an entry point for further attacks on critical system resources, networks or database servers.

The research described here examines the possible security threats to software programs developed in Java. Specifically, the vulnerabilities in Java bytecode are assessed. There are two major reasons for choosing Java bytecode as the subject of this vulnerability assessment. First, industry is using Java as one of the two major platforms for development (the other is .NET architecture). Second, Java bytecode carries more information then native executable code, and thus opens the door for possible exploitations. In order to attack a Java application, a hacker can exploit the vulnerabilities in Java bytecode. This thesis addresses two main issues: *"What can one exploit?"* and *"How can it be exploited?"* concerning Java bytecode.

The primary focus of this thesis is on assessing the vulnerabilities of Java applications developed in a J2SE environment. This assessment examines the class files with Java bytecode instructions. The research focuses initially on different classes of software vulnerabilities, such as intrusion penetration, component penetration, and intellectual property penetration [Umphress 2004]. The associated literature survey concentrates on documentation of the JVM specifications [Lindholm and Yellin 1999], class file format, and the different types of exploitations that are possible by looking at bytecode. The thesis illustrates issues such as possible security threats due to reverse engineering, replacing and patching Java classes, decompilers, obfuscation techniques and other issues. A taxonomy of the bytecode vulnerabilities is developed followed by a case study of a real life Java application. The case study involves an extensive vulnerability assessment of the application and its components. The assessment results

have been validated in the concluding section. Possible solutions and ways to protect against such bytecode exploitations are proposed. The research groups and corporate industries engaged in development and maintaining Java applications will thus greatly benefit from the work reported here.

CHAPTER 2 LITERATURE REVIEW

2.1 Security and Software Vulnerabilities:

With the advent of information technology, the routine use of many different types of software applications has become part of our everyday life. Complex software programs and applications are widely used in critical systems, throughout the medical, finance, business and research sectors. However, protecting these applications has become a challenging task for researchers and developers. It is no longer possible to ignore the security implications of a simple stand-alone program, because even though software applications do not always contain sensitive data, there is always the possibility that they can be broken into, scrutinized, modified or exploited in many harmful ways [Umphress 2004].

Software applications have always attracted the hacker community to execute illegal and frequently destructive activities. A software application can become an entry point for an unauthorized user if rigorously exploited with available resources and time. Once a malicious user has gained control over the application, he or she can get hold of all the underlying functionalities, secret algorithms, data structures, functions and methods. If the application is decomposed by reverse engineering the design and exploiting the entry point, it becomes possible to introduce malicious code to harm the application itself, along with critical system and application resources. A smart hacker

with good tools and the latest technology can exploit the software vulnerabilities in many different ways. Software vulnerabilities may also be exploited in order to gain unauthorized access to the system, which can then be used for further destruction.

2.1.1 Classification of Software Vulnerabilities

Software vulnerabilities may be classified in three ways, depending on the ultimate intentions of the attack.

2.1.1.1 *Intrusion penetration* [Umphress 2004]*:* The word "intrusion" is defined as accessing something without the owner's permission. In the case of software, "intrusion penetration" signifies gaining control of the software application by masquerading as an authorized user. The ultimate intention of intrusion penetration is to learn protected information, such as how the application is authorizing its users, how the application is authenticating the data, knowing the encrypting algorithms that the application is using, knowing and possibly exploiting the data validation functions and decrypting algorithms.

2.1.1.2 *Component penetration* [Umphress 2004]: "Component penetration" is the most common and comparatively easily exploitable penetration because most software applications are component-based or module-based and thus are easily decomposed. Component penetration begins by decomposing the application's functionality or flow in different contexts or components, allowing the attacker to understand the flow of the program and view each part of the system individually. The attacker can then use the important underlying functionalities in order to assemble competitive software rather than having to build it from scratch. Alternatively, the attacker's objective may be to replace

or patch the original component with a component having the same interface but designed to exploit the application or the system resources.

2.1.1.3 *Intellectual penetration* [Umphress 2004]: The ultimate intention of "intellectual penetration" is to expose hidden information, such as data structures, classified information, business rules, and functions that manipulate sensitive data, such as customers' social security numbers, bank account information etc.

2.2 Java

Java was first introduced by Sun Microsystems [Sun Microsystems Inc. 1996]. It was designed to be a portable and secure language for web development. Java security includes language features such as array index range checks, bytecode verification, controlled access, an automatic garbage collection system, and sophisticated access control mechanisms built over stack inspection techniques [Kalinovsky 2004].

Java is one of two primary development architectures which are used by corporations and researchers for different purposes. .NET is the other powerful architecture that can provide seamless integration of computing and communication resources provided for different types of architectures over internet and other multi-tier architectures.

2.2.1 Java source code and Java byte code

An installable Java program does not contain source code but consists of Java class files with bytecode instructions. The bytecode is composed of a stream of bytes with a specific format. The class file format is specified by the Java Virtual Machine

(JVM) specification [Lindholm and Yellin 1999] which will be discussed in more detail later in this literature review. The bytecode looks like a collection of assembly language instructions when viewed by a disassembler.

Unlike Java, which compiles the source code into intermediate bytecode, other traditional programming languages like C and C++ code are directly compiled into native machine language, which is very difficult to understand and exploit because it retains less information than the Java bytecode. Java bytecode retains most of the information of the source code because it was designed for platform independence, portability and network mobility. This is what makes the Java bytecode more vulnerable. If proper precautions are not taken by developers, Java bytecode can reveal almost all the information that one can know by reading source code, such as class, method and variable names, control flow of the program, data structures, sensitive algorithms and functions. It is possible to reverse engineer the Java class file using many of the commercially available reverse engineering tools and then manipulate the underlying logic of the Java application. These reverse engineering tools and techniques, along with ways to minimize and secure Java code, will be examined in more detail later in this literature review.

Figure 2.1 shows the compilation and execution sequence of a typical Java application, and displays how a hacker carries out the three types of penetration attacks on the class file.

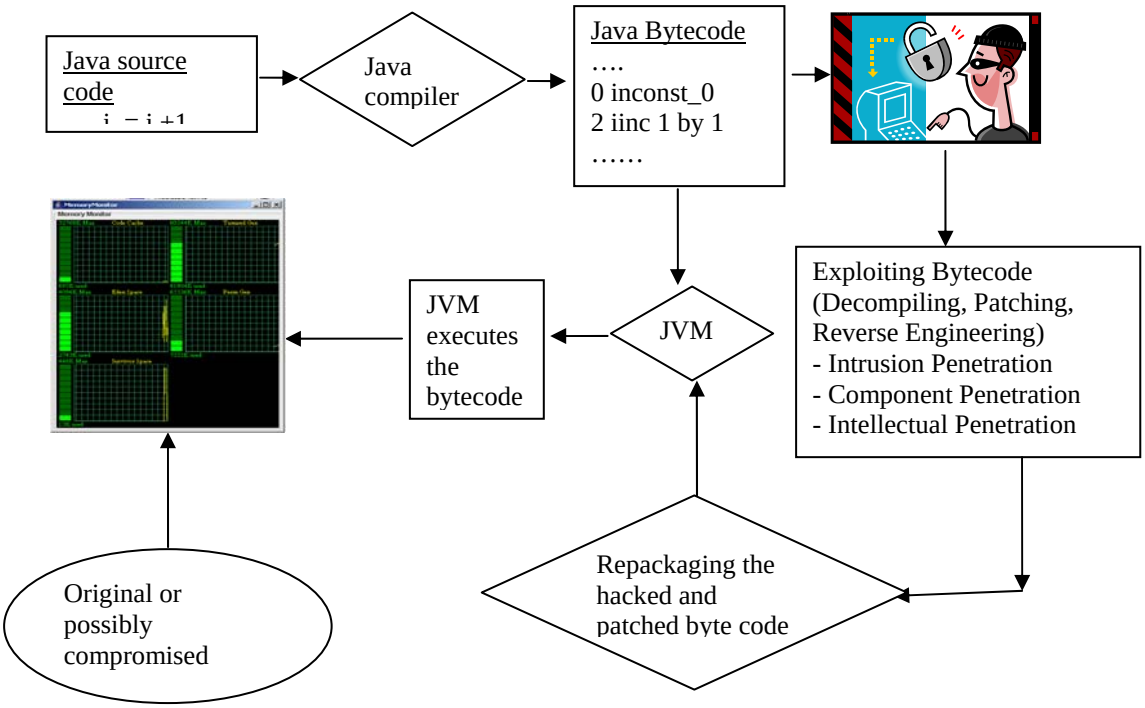

Figure 2.1: A Possible Scenario for Hacking a Java Application

2.2.2 Java Class File

The source code of any Java program is compiled into an intermediate binary format (bytecode) with a .class extension. The class file format is defined by the JVM specifications [Lindholm and Yellin 1999]. This class file format is standard across all platforms it is similar for all Java compilers. A typical class file contains a series of data structures representing the class or interface itself, methods, fields, and attributes. Each class file contains the complete description of a single class or interface. The detailed review on the class file format is discussed in the Appendix H.

When one looks at the bytecode in any traditional HEX editor, it consists simply of a stream of hexadecimal numbers. However, the disassembled bytecode is more readable and can be represented by their mnemonics as shown in the Figure 2.2.

```
// Bytecode stream: 03 3b 84 00 01 1a……. [Venners  1999]

// Disassembly:
i       const_0             //03
i       store_0             //3b
i       inc 0, 1            //84  00  01
i       load_0              //1a
                            ……..
```

Figure 2.2: Sample bytecode stream and its corresponding assembly-type format

2.3 Revealing vulnerabilities by Exploiting Java bytecode

The systematic and well-defined structure of the Java bytecode can be very vulnerable at times. Below is a listing of some possible Java bytecode exploitations that can fall under one or more classes of the software vulnerabilities discussed earlier:

- *Revealing the names of the classes, their methods, and local and method variables*. From the point of view of normal users and developers, it may appear totally innocent if these names are exposed. However, exposing these names may provide a starting point for a penetration attack. Certain names of classes and functions may attract the hacker. Suppose, for example, that the name of a class is “userAuthentication” or the name of a method is “checkPassword”. These names will quickly attract the attention of a possible intruder for further exploitation.
- *Reverse engineering the bytecode*. This generates the source code, which a smart hacker can use to substitute any original function with a malicious function and then repackage the application. Using this malicious code, the hacker can gain control over the system on which the program is running and can further exploit the system.

- *Software piracy*. This is one of the major threats facing the software industry. A cracker can obtain the copyright information and then remove the propriety watermarks before repackaging and selling the software.
- *Understanding the importance of components from the bytecode*. This assists other vendors to extract and re-use the code. Obtaining cryptographic algorithms and other critical functionalities of financial institutions pose an obvious threat to data security, in addition to fraudulent use by other vendors.
- *Cracking the Java bytecode and extracting the function that performs user authentication*. Once hackers understand the logic behind an authentication process, they can substitute or tamper with that functionality in order to bypass the authentication step.
- Software applications may not themselves contain the data but they do process it. Thoroughly *exploiting bytecode can help hackers to understand the data processing*.
- *Information on the classes and their hierarchy*. This can help hackers to systematically understand the flow of the program by decomposing the application, thus giving hackers the access to the code they are seeking.
- *How the program is structured.* It is possible to discover the internal working of a program or learn about the implementation of special features or algorithms, coding techniques, and sensitive information by exploiting the Java bytecode.
- *Knowledge of internal data structures*. It is also possible to discover information about the data structure, and the functions that manipulate and authenticate the database by exploiting vulnerabilities in Java bytecode, making it possible to change the way those functions carry out these functionalities for further exploitation.

- *Access to the program's internal elements.* A hacker can change the values of internal variables, condition checking, pop-up and text messages, user-interfacing, color schemes, and visual elements of the program.

2.4 Decompilers and Reverse Engineering.

Reverse engineering is defined as *"Analyzing a subject to identify its current components and their dependencies and to extract and create a system abstraction and design information"* [Suryadevara and Ahmed 2004]. This technique has gained ground in today's world of information and business technology due to the increasing demands of changing legacy systems into new multi-tier architectures in time and cost effective ways. Reverse engineering has opened up major opportunities for analyzing the original code in the absence of either documentation or source code, understanding sub-system decomposition, internal design patterns, program slicing and dicing, dynamic and static program dependencies, object-oriented metrics and a great deal more. However, this powerful technology is not used only for the beneficial purposes stated above, but has also been adopted by the hacker community in order to break and tweak software.

Decompilers are one of the tools commonly used for carrying out this type of reverse engineering. Their operation is exactly opposite to that of compilers. A compiler transforms the source code to machine readable or intermediate code, whereas decompilers re-transform the intermediate code (byte code in Java, MSIL in .NET) into something closer to the original source code. This technique increases the possibilities for exploiting any vulnerability.

There are many decompilers that are commercially available with which one can carry out reverse engineering attacks, the best known of which are JAD [Jad 1997], JODE [Jode 1998], and Mocha [Mocha]. Decompilers provide the leverage whereby one can understand the internal logic and change the program's structure and code that may affect the ultimate functionality of the program. This is what is known as "patching" in the developer community.

2.5 Obfuscation

Obfuscating is the technique used to transform bytecode in order to make it harder to understand after decompilation. It incorporates various techniques, such as replacing the names of the classes, parameters, packages, and variables with machine-generated names, and removing the debugging information from the source code [Kalinovsky 2004]. Some advanced obfuscators can even change the control flow of a Java program by inserting bogus code within the original code. These techniques will certainly not prevent a hacker from reverse engineering the bytecode, but can at least make his task harder. However, obfuscation has many drawbacks associated with it. For example, it may decrease the overall performance of the application. Renaming certain packages may affect how the API accesses those packages and thus affects the working of the application. The research reported here presents this and other techniques that may be used to protect the Java bytecode from successful attacks.

CHAPTER 3 TAXON0MY OF JAVA BYTECODE VULNERABILITIES

Software is a form of digital data, and thus is vulnerable to theft and misuse. *Software vulnerability* can be defined as a security hole or flaw that can be exploited by a malicious user for illegal purposes with the intent to damage, gain unauthorized access, access information that was intended to be hidden, or carry out other types of attacks on the software. The adverse effects can include the loss of business revenue, damage to the reputation of the software vendors, or leakage of sensitive classified data.

Software vulnerability attacks (SVA) [Umphress 2004] is the process of assessing and analyzing software to detect potential vulnerabilities. The primary focus of this research is to carry out a software vulnerability assessment on Java bytecode. An increased understanding of software vulnerability and ways to prevent and eliminate it can be achieved by developing a generic taxonomy of software vulnerability. The new taxonomy classifies software vulnerability in terms of the nature or ultimate intent of the attack, ways of carrying out that attack, severity of the vulnerability, and possible mechanisms or suggestions to prevent or eliminate that vulnerability. This taxonomy is developed using Java bytecode as the subject of this software vulnerability assessment.

3.1 Prior Work on Vulnerability Taxonomy Development and Classification

A taxonomy is a system of classification, including its principles, procedures, and rules [WEBOL 1998, Simpson 1995]. The goal of this taxonomy development is to

propose a mechanism or possible ways of detecting or preventing specific types of attacks which could be carried out by exploiting underlying vulnerabilities in Java bytecode.

Several research projects have been carried out in order to classify software vulnerability [Bishop 1999]. However, the previous attempts of vulnerability classifications were more abstract in nature and did not concentrate on a single platform or programming language. Therefore, it is always problematic to apply a single classification or model to a specific software platform or programming language for a vulnerability assessment. The discussion below will allow us to review the previous work on vulnerability classification, as well as allow us to define the common characteristics and parameters that are required to develop a comprehensive taxonomy. There is a large amount of literature concerning the threats on software and information security. A review of prior work on security faults and vulnerability classification, along with development of a new taxonomy, will facilitate this software vulnerability assessment of Java bytecode.

3.1.1 RISOS [Abbott et al. 1976]

The RISOS (Research Into Secure Operating Systems) project was designed to identify the common security flaws in operating systems and to suggest possible operating system security enhancements. A list of possible security flaws was developed and the flaws were classified based upon the time the flaw was introduced into that system, or the section of code it was introduced.

The RISOS study defined seven classes of security flaws:

- Incomplete parameter validation.
- Inconsistent parameter validation.
- Incomplete sharing of privileged data.
- Asynchronous validation.
- Inadequate identification/authentication.
- Violable prohibition/limit.
- Exploitable logical error.

3.1.2 Cheswik and Bellowin [1994] Classification

For their study on firewalls, Cheswik and Bellowin classified the attacks into seven as specified below:

- Stealing password.
- Social engineering.
- Bugs and backdoors.
- Authentication failure.
- Protocol failure.
- Information Leakage.
- Denial-of-Service

3.1.3 Incident taxonomy [Longstaff and Howard 1998]

Longstaff and Howard presented a process-based incident taxonomy for computer and network attacks. Their approach considered the factors such as the motivation and objectives of attacks. The Figure 3.1 shows the taxonomy they developed, along with a

classification of each type of incident. Their taxonomy consists of five different stages: tool, vulnerability, action, target, and unauthorized results.

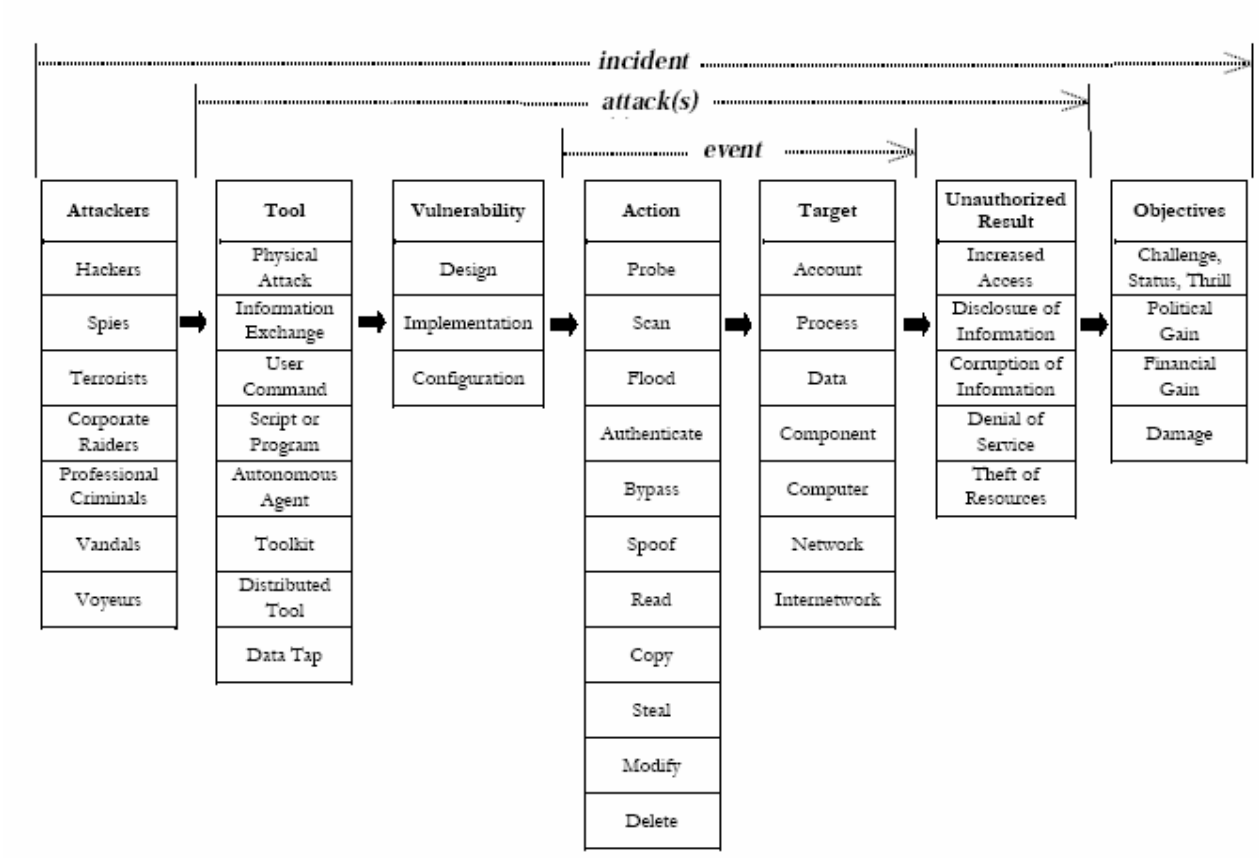

Figure 3.1: Howard's Taxonomy. [Longstaff and Howard 1998]:

3.1.4 Ultimate intent classification [Umphress 2004]

Umphress classified software vulnerabilities into three categories based on the ultimate intention of the attack:

- Intrusion penetration: The ultimate intention of this kind of penetration would be to gain access to the software application by masquerading as a legal user and further exploit that application.

- Component penetration: Component penetration begins by decomposing the subject software application into different modules or components. The attacker then can view each part of the application as separate entity and understand the flow of program for further exploitations.
- Intellectual penetration: The ultimate intention of intellectual penetration would to obtain hidden information such as classified or valuable cryptographic algorithms, business rules, data validation schemes, etc.

3.1.5 Threat classification [Power 1996]

The threat to the subject software application or system that a vulnerability poses was classified into four main categories [Power 1996], namely threat to integrity (modification), threat to authenticity (fabrication), threat to confidentiality (interception), and threat to availability (interruption). A particular attack or software vulnerability can seldom exactly be classified into one of the above categories, but generally poses one or more classes of threats.

The previous studies on vulnerability classification helped to identify the various parameters of software vulnerability which should be considered during the development of a new taxonomy in the current research. Some of the parameters of interest include the origin of the vulnerability, threat caused by that vulnerability, ultimate intention of the attacker, severity of the attack, etc. Two of the prior discussed classification models provide the best fit for a comprehensive classification of the identified flaws in Java bytecode, namely ‘Threat classification’ [Power 1996] and ‘Ultimate intention classification’ [Umphress 2004].

3.2 Development of Java Bytecode Taxonomy

The taxonomy developed during this research takes into account the significance of each class of attack, the categorization of the identified vulnerabilities into the two classification models discussed above, the setting or scenario in which the attack is possible, and the techniques used to carry out the attack. The problem of interest is to classify the vulnerabilities in Java bytecode for a stand-alone program, revealing as many as security loop holes and vulnerabilities in the Java bytecode.

The following steps were followed during the development of the software vulnerability taxonomy:

- Identify a rich set of possible attacks or the different classes of possible exploitation of Java bytecode.
- Support the identified classes of possible vulnerabilities with an example or scenario of a possible attack, how the attack can be carried out, etc.
- Classify the identified software vulnerabilities into three categories, based on the ultimate intent of the attack or software penetration, namely intrusion penetration, component penetration, and intelligent penetration. It may not be possible to categorize the identified vulnerabilities exactly into one category and it may fall under one or more categories, mainly because these categories are not mutually exclusive to each other and a single attack on any Java application can be carried for multiple intentions.
- Classify the software vulnerabilities based on the ultimate damage [Power 1996] it can cause to the subject application, such as an attack on th integrity (modification), authenticity (fabrication), confidentiality (interception), or availability (interruption) of

the subject application. Thus, each vulnerability can fall under one or more of the three categories and can cause one of four types of damage to the Java program.

3. 3 Classes of of Java Bytecode Exploitations

3.3.1 Identifying the classes of vulnerabilities

1) The names of various key elements of any Java program, such as the package, classes, super-classes, interfaces, methods, and local and class variables, can be revealed.
2) The signatures of class methods can be revealed.
3) Class hierarchies and class dependencies can be revealed by exploiting Java bytecode.
4) The copyright information or propriety watermark of a Java application can be hacked and removed for piracy purposes.
5) A program's internal elements, such as its pop-up windows, messages and alerts, user-interfacing color schemes, or visual elements, can be hacked.
6) Java bytecode can be reverse-engineered to generate source code using various decompiling tools.
7) It is possible to discover the internal working of the program or learn about the implementation of special features or algorithms, coding techniques, and sensitive information by exploiting the Java bytecode.
8) Data validation schemes or data processing can be revealed by comprehensively exploiting the Java class file.

9) Bytecode can systematically be instrumented in order to introduce new logic for further exploitation. Thus, some internal functionality or values of local variables can be altered during the attack (patching).

3.3.2 Discussion and classification of identified classes of exploitations.

The previous section summarized a set of possible exploitations of Java bytecode. These vulnerabilities need to be classified in order to develop a generic taxonomy. An explanation of the significance of each type of attack and a discussion of an example or scenario for each attack will facilitate this classification.

1) Revealing the names of key elements of Java programs.

   a) Significance: This can be the first level of attack and is possibly the simplest to accomplish. Large Java applications may contain as many as 500-600 classes, but the hacker will be interested only in specific classes or method implementations. Generally, the names of classes, methods, and variables are given logical names, since giving logical names also helps to maintain the application. However, once the names of classes, methods, or variables are revealed to the hacker, these can be used for further exploitation.

   b) Example: Suppose a malicious user is interested in hacking the authentication functionality of the program. In such a case, the hacker would search the Java class files for the particular class or method declaration. Class or method names that would attract the attacker's attention would include "*UserAuthentication*", "*checkPassword*()", or "*authorizeUser*()" etc. Thus, if the class elements names are

easily revealed, then this can be dangerous as far as the security of the program is concerned.

c) Classification: The ultimate intention of this class of attack is likely to gain unauthorized access to the program by knowing and changing the functionality that authorizes the user (e.g. the '*userAuthetication()*' function). The attacker may be seeking to identify the important classes that incoporate cryptographic algorithms or process sensitive customer information such as bank account numbers or social security numbers. Therefore, this type of exploitation can be classified as "intellectual penetration" as well as into "intrusion penetration". Furthermore, this class of vulnerability causes a major threat to the confidentiality (interception) and integrity (modification) of the attacked Java program. Interception occurs when an unauthorized user gains access to the application by knowing the names of elements of the class file. Once these names are revealed to the unauthorized user, he or she can tamper with the sensitive functionality, opening the way for further exploitations, and causing a threat to the integrity of the Java program.

2) The signatures of class methods can be unveiled.

a) Significance: Exposure of the exact signature of the methods that process sensitive data to the unauthorized user can be the second level of attack. Sometimes, simply knowing the class or method names are not enough for attacking a Java program. The hacker may also be required to know the method signatures if he or she seeks to augment the existing method or instrument it by inserting new functionality. Method signatures contain the data types of arguments, the data type of return value,

and the access levels. Knowledge of a method's access level can disclose the scope of that method. Method names and their signatures can further unveil the overloaded methods with the same names.

b) Example: Consider the following overloaded methods' signatures that print different results based on the object passed to it at run-time.

- *void print (BankAcc)*
- *void print (CustomerClass)*

If these signatures are exposed by the attacker, then he or she will discover that there may be two methods with the same name "*print*()" having different functionalities. One method takes an object of class *BankAcc* and prints the details of bank accounts, whereas another print method takes an object of *CustomerClass* as an argument and prints details for that customer. Thus, the hacker can choose which method he or she wants to exploit further. Hence, knowledge of the exact signature can expedite further attacks.

c) Classification: The ultimate intention of knowing the signatures and access level of methods can be to gain unauthorized access to the program or to understand the functionality that authenticates the users. Once the hacker knows that method's signature, he or she can work around that functionality to obtain unauthorized access. Revealed signatures of other important methods that process sensitive data can also be harmful. Thus, this class of exploitation can be classified into "intrusion penetration" as well as "intellectual penetration". This kind of attack poses a threat to the confidentiality as well as integrity of the Java program.

3) Class hierarchy and dependency can be developed by exploiting Java bytecode.

a) Significance: Knowledge of the overall class hierarchy and class dependency, sub-system decomposition, internal design patterns, understanding dynamic and static program dependencies, etc. can be helpful to an attacker while exploiting Java applications. Basic dependency and class hierarchy is important for understanding object-oriented software applications [Barowski and Cross 2002]. Discovering the individual components can exhibit the overall structure of the whole application. A class diagram can help a hacker to understand the dependencies amongst the different components, which in turn can facilitate the chance of finding an entry point and navigating the execution sequence until the hacker finds the component or functionality of interest. Component decomposition allows a particular component with an important algorithm or functionality to be re-used in another program. A hacker can extract and replace a program component with his or her own version of that component incorporating malicious code that can have the same interface with different functionality. The inserted component can report the inner working of the application, thus giving further exploitation opportunities to the intruder [Umphress 2004].

b) Example: Consider the class hierarchy shown in the Figure 3.2 revealed by the hacker.

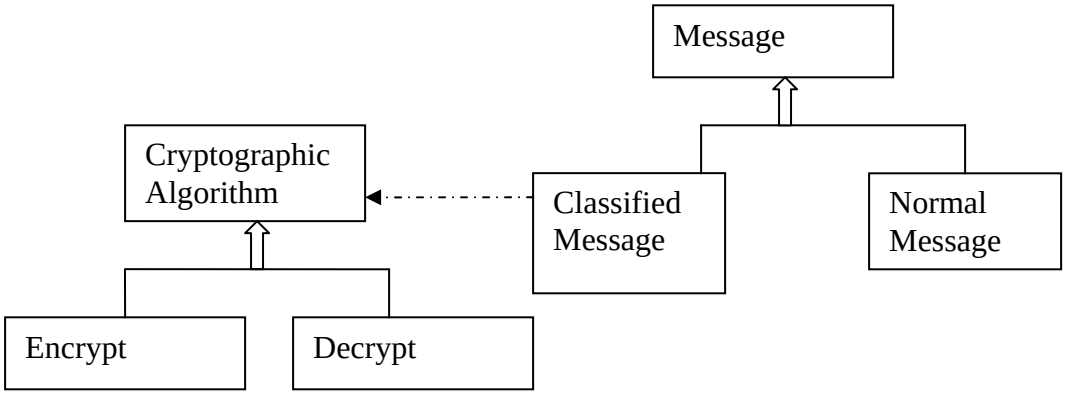

Figure 3.2: A Class diagram of a program with cryptographic functionality.

If the hacker gains access to the *Cryptographic Algorithm* class, he or she can re-use it in a new program without needing to write the whole algorithm from scratch, or substitute this component with another compromised cryptographic algorithm.

c) Classification: This class of attack can be categorized as "*component penetration*", because the hacker's primary intention is to decompose the entire program into separate visible components in order to reuse one of the important components or substitute that component with another corrupted component containing malicious code. This type of possible component exploitation is a threat to the integrity of the program as the hacker can subvert any component for illicit use. If any component is substituted by another malicious component, then it is a threat to the authenticity of the program.

4) The copyright information or propriety watermark of Java application can be pirated.

a) Significance: Software piracy is one of the greatest threats faced by commercial vendors developing licensed software applications. Different watermarking techniques are implemented in order to prove the ownership of software or even the data structures or algorithms used in the software. Watermarked software can be attacked with the objective of locating, altering, or removing the watermark. The degree of resistant to attack determines the quality of watermark. A attacker can systematically exploit class files of Java programs and remove copyright watermark information before making pirated copies for private use, or reselling the software. A hacker can discover the section of code that includes embedded information about the copyright information or the customer identification number.

b) Example: Suppose the following is a snippet of source code showing a class definition that includes the future expiration date of the trial version of a program. A hacker can gain access to this part of code by exploiting the Java bytecode, since class the file retains almost all the information of the original source code.

```
Public class LicenceManager {
        Private string host;
        Private string ipAddress;
        Private Date expires;
        ……..
}
```

Pseudo Code Segment 3.1: Example of Bytecode Vulnerability Class IV

c) Classification: The attacker's objective in cracking the copyright information would be to make pirated copies for illegal use. So, this class of attack can be classified as "intellectual penetration". This vulnerability poses a threat to the

software program's confidentiality (interception). Removal of watermarks can also make the program unusable in some cases. Thus, it also poses threat to the software's availability (interruption).

5) Internal elements of the program, such as pop-up windows, messages and alerts, user-interfacing color schemes, and visual elements can be hacked.

a) Significance**:** A typical Java application may contain various user interface elements such as alerts, text messages, icons, menus, or pop-up windows. Java class files include corresponding bytecode that controls the visual layout, such as menu composition, color schemes, etc. A hacker can access the section of code that displays alerts or text messages, or controls the user interface elements and color schemes by using sophisticated reverse engineering and decompiling techniques and tools. Although code obfuscation can somewhat prevent reverse engineering techniques, with sufficient effort an attacker can locate the code that deals with the internal elements, and then alter the text messages, alerts, or color schemes. These kinds of attacks are generally not very damaging, but can be very annoying for the user who is using the patched applications.

b) Example**:** Consider the following pseudo code that displays a pop-up window asking the user whether he or she wants to save the current changes:

If (*changes are made*)
    Display pop-up window "*Do you want to save the changes? – Yes, No, Cancel*"
End If

Pseudo Code Segment 3.2: Original Pop-Up Window Message

Any hacker can locate the part of code that displays this pop-up window using standard search tools or by running a binary search on the directory containing class files and other configuration files deployed with the application. Using routine patching techniques, the hacker can replace the above message with another message which is either not correct or totally absurd, as shown below:

If (*Changes are made*)
Display pop-up window: *"Do you want to save the changes? – "**$#%, ###, &***^^"
End If

Pseudo Code Segment 3.3: Hacked Pop-Up Window Message

c) Classification**:** generally such kinds of attacks are not carried out with the intention of gaining control of the program by masquerading as a legible user or extracting and re-using any component of the application by decomposing it. However, this class of attack possesses a threat to the user interface elements of the program. So, this class of vulnerability can be classified as "intellectual penetration". This vulnerability poses threat to the integrity (modification) of the application's user interface elements. Since the hacker can also remove important alerts and text messages by modifying and repackaging the class files, there is an inherit threat to the availability (interruption) of such internal elements.

6) Java bytecode can be reverse-engineered using various decompiling tools in order to generate source code.

a) Significance: Java source code is compiled into bytecode, which is then interpreted by JVM. The Java bytecode is very susceptible to reverse-engineering because it adheres to well-defined JVM specifications and there is almost a one-to-

one relationship between the original source code and the bytecode. Many tools may be used for reverse engineering, including disassemblers, decompilers, fault injection tools, etc. Various techniques have been proposed to tackle this class of threat, such as obfuscation, cryptographic techniques, watermarking etc.

A hacker can exploit the reverse engineering vulnerability of Java bytecode for many malicious objectives. Reverse engineering allows users to learn about a program's internal structure and logic, along with intellectual property included in the application, such as important algorithms and functionalities. Reverse engineering techniques are capable of analyzing code, system decomposition, analysis of static and dynamic program dependencies. This class of software vulnerability has an overlapping with some other vulnerability classes. In other words, many other types of attacks are carried out by exploiting this vulnerability. For example, identifying the class hierarchy and dependency, software piracy, hacking a program's internal elements, and learning the internal logic of the program can be achieved by reverse engineering the Java bytecode.

b) Example:

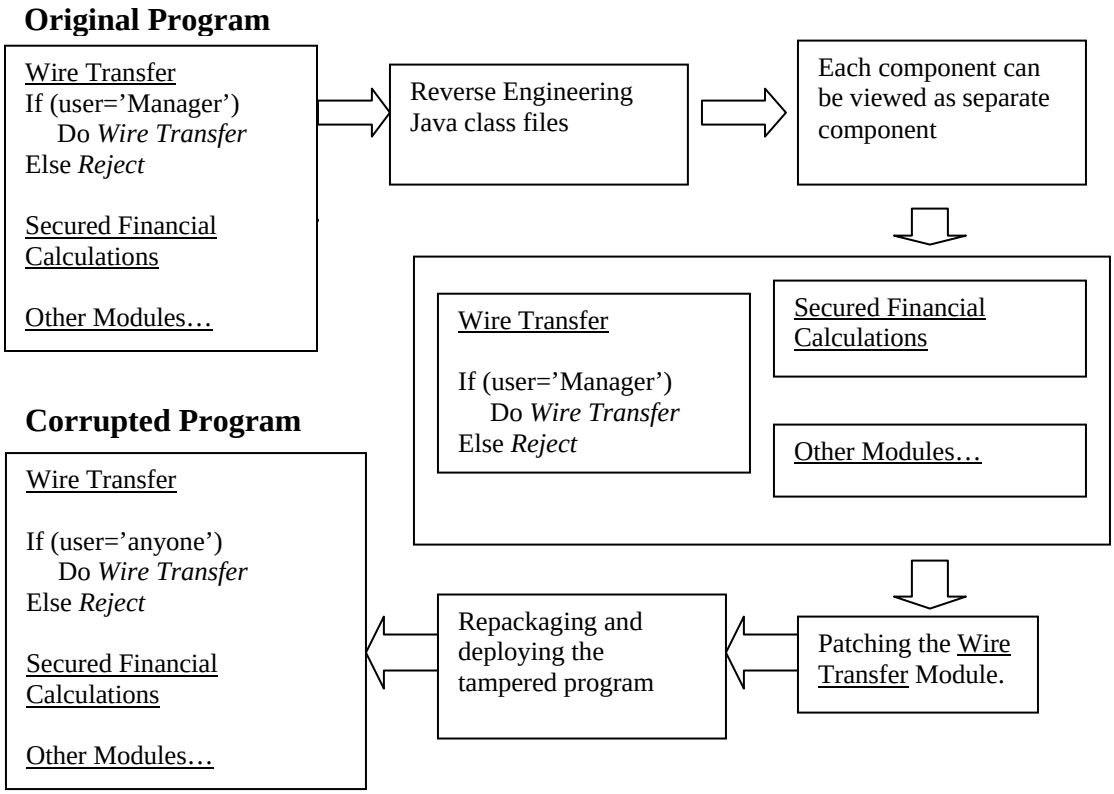

Figure 3.3: Scenario depicting Reverse Engineering attack

In the above scenario, the original program has a module that authorizes *wire transfer* transactions. If the user is *Bank Manager* then the transaction is carried out, otherwise the request is rejected. The other module includes a collection of *sensitive financial calculations* which are the intellectual property of the application. The program has other components in addition to these two modules. Any malicious user will be able to retrieve the source code by carrying out a reverse engineering attack on the class files. Now he or she can view all the components of the program as separate entities. The attacker can then locate the code which authenticates the user as *Bank Manager* before the *wire transfer* is carried out. The patching is carried out by replacing *Bank Manager* with *anyone,* thus skipping the authentication phase. Now the tampered version of the

program is repackaged and deployed. This allows the hacker to carry out illegal wire transfer transactions.

c) Classification**:** Reverse engineering attacks can be carried out with the intention of gaining un-authorized access to the program, decomposing the program into separate modules, or discovering business rules, classified data, or secret algorithms. Thus, a reverse engineering attacker might have a wide range of intentions and this type of attack can be categorized into “intrusion penetration”, “component penetration”, and “intellectual penetration”. Reverse engineering allows a hacker to tamper with the original functionality of the program, thus posing a threat to its integrity (modification). It is also possible to gain acquire the intellectual and secret information by reverse engineering the bytecode, thus posing a threat to the confidentiality (interception) of the target application.

7) It is possible to discover the internal working of the program or learn about the implementation of special features or algorithms, coding techniques, and sensitive information by exploiting the Java bytecode.

a) Significance**:** Java class files strictly adhere to JVM specifications and have a very well-defined format. Class file has bytecode which retains almost all the information from the source code. This makes Java bytecode vulnerable to this kind of attack, where hackers seek to discover the internal working of the program, its important functionalities, ideas behind the code, secret algorithms, and the other intellectual properties contained in the program. Applications may contain business rules, protected financial calculations, or functions that process or manipulate

sensitive and classified data. Many techniques are available to protect the intellectual property of the program, such as obfuscation, copyrighting code, watermarking, cryptography etc. Bad coding practice, weak obfuscation techniques, and reverse engineering tools all allow hackers to exploit this class of the vulnerability.

b) Example**:** Consider the following scenario,where the sender's application sends an email by encrypting it and the receiver's application decrypts it. The encryption and decryption functions are intelligent properties of the program. If a hacker gets hold of these algorithms, then he or she can decrypt any incoming email message.

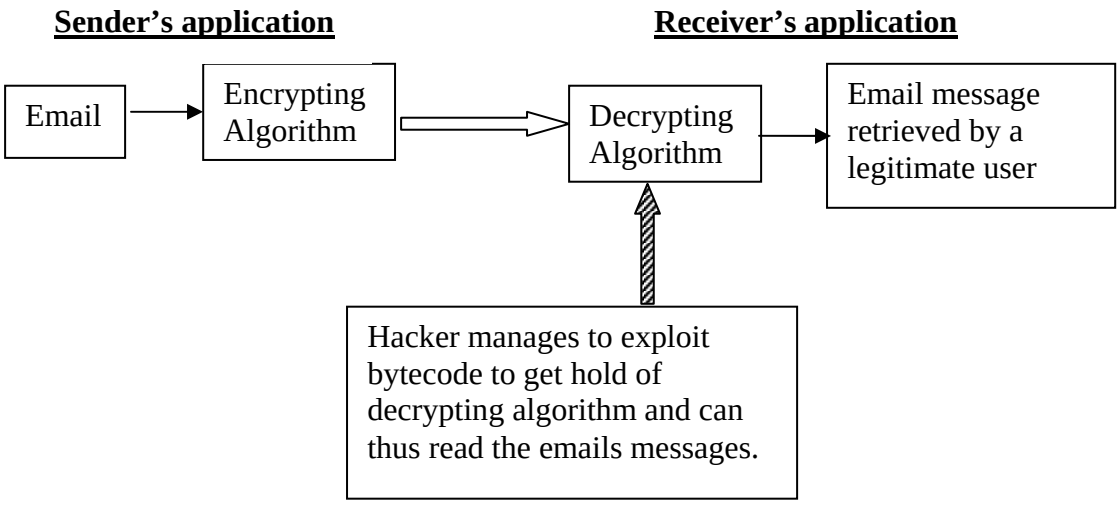

Figure 3.4: Hacker manages to read encrypted messages by discovering the decryption algorithm

c) Classification**:** The ultimate intention of a hacker seeking to exploit this class of vulnerability is to gain access to intelligent and confidential assets of the application. Thus, this can be classified as "intellectual penetration", with an inherit threat to the confidentiality (interception) of the program and its contents.

8) Data validation schemes and data processing functions can be revealed by thoroughly exploiting the Java class file.

a) Significance**:** Java applications may not contain actual data, but always include functions that process, manipulate, or authenticate the data. The platform independent and portable nature of Java applications requires Java class files to retain almost all the information of source code, which makes Java bytecode more vulnerable to possible exploitation. Thus, Java class file should have all the information on data types, data structures, and functions that work around the data. Hackers can change the functions that authenticate and manipulate the data, insert new logic, and thus further exploit the program.

b) Example**:** Consider a Java application used by a financial institution to process a database of customers' SSN, password, bank account numbers, etc. The application contains functions that accept the SSN and password as input and then query the database to authenticate the user. If a hacker is able to locate the code which authenticates the user, he or she can insert or change the logic to skip the authentication step.

c) Classification**:** The ultimate intention of the hacker in exploiting this class of vulnerability is to gain unauthorized access to the program or to reveal the functions that work around the database. Thus, this kind of attack can be classified as "intrusion penetration" or "intellectual penetration". According to the threat classification [Power 1996] scheme, this class of exploitation damages the application by attacking its integrity (modification) and confidentiality (interception).

9) Bytecode can systematically be instrumented to introduce new logic for further exploitations. Thus, some internal functionality or values of local variable may be altered during the attack (patching).

a) Significance**:** It is possible to work at the bytecode level to instrument existing class files to introduce new logic and programmatically generate new classes for further exploitation. This can be done by manually hacking the bytecode or using various tools and techniques. For example, the open source library from Apache known as the Byte Code Engineering Library (BCEL) [BCEL 2003] is a tool that allows user to analyze, create and manipulate Java class files.

Instrumenting means inserting new bytecode or augmenting existing class files [Kalinovsky 2004]. This class of attack allows a hacker to first review the bytecode and locate the target function, then change the way the function behaves by adding new logic or changing the value of local variables. Once the hacker locates the function or variable to be tweaked, he or she has to alter it at bytecode level, and then repackage the tampered class file.

b) Example**:** Consider the following snippet of code which verifies whether the length of password entered by a user is more then eight characters long. An attacker can alter this to bypass the minimum password length verification, which can be done at the bytecode level.

```
minLength int;
minLength = 8;

If (password.length() <= minLength)
{
Then
System.print.out("Password length should be greater then
                                        eight characters");
}
Else
        Proceed…..
```

Pseudo Code Segment 3.4: Password Length Validating Code

If the hacker is successful in changing the value of the variable *minLength* from 8 to 0, then the new logic would allow him to skip the minimum length checking of the password. This is possible if the server side checking is not implemented.

c) Classification: This kind of attack is carried out to gain unauthorized access to the program and can be classified as "intrusion penetration". Instrumenting the existing bytecode can be carried out with the intention of decomposing the application and substituting with another one that has new logic and executes the malicious functionality. Thus, this can also be classified as "component penetration". This class of the vulnerability poses a threat to the integrity of the target application.

3.4 Summary for the Taxonomy of the Bytecode Vulnerabilities

Table 3.1 summarizes the ultimate intention classifications for each class of the Java bytecode exploitations.

	Ultimate Intention Classification [Umphress 2004]		
Classes of Java bytecode exploitation	**Intrusion Penetration**	**Component Penetration**	**Intellectual Penetration**
Revealing the names of key elements such as class, super-class, interface, methods, and variables.	●		●
The signatures of class methods can be revealed.	●		●
Class hierarchy and dependency can be developed by exploiting Java bytecode		●	
The copyright information or propriety watermark of Java application's can be hacked and removed for piracy purpose.			●
A program's internal elements such as pop-up windows, messages and alerts, user-interfacing color schemes, visual elements can be hacked.			●
Java bytecode can be reverse-engineered to generate source code using various decompiling tools.	●	●	●
It is possible to discover the internal working of the program or learn about the implementation of special features or algorithms, coding techniques, and sensitive information by exploiting the Java bytecode.			●
Data validation schemes or data processing can be revealed by thoroughly exploiting the class file.	●		
Bytecode can systematically be instrumented to introduce new logic for further exploitation. Thus, some internal functionality or values of local variable's can be altered during the attack (patching).	●	●	

Table 3.1: Ultimate Intention Classifications for the Java Bytecode Vulnerabilities

The table 3.2 summarizes the ultimate intention classifications for each class of the Java bytecode exploitations.

	Threat Classification [Power 1996]			
Classes of Java bytecode exploitation	**Modification**	**Fabrication**	**Interception**	**Interruption**
Revealing the names of key elements such as class, super-class, interface, methods, and variables.	●		●	
The signatures of class methods can be revealed.	●		●	
Class hierarchy and dependency can be developed by exploiting Java bytecode	●	●		
The copyright information or propriety watermark of Java application's can be hacked and removed for piracy purpose.			●	●
A program's internal elements such as pop-up windows, messages and alerts, user-interfacing color schemes, and visual elements can be hacked.	●			●
Java bytecode can be reverse-engineered to generate source code using various decompiling tools.	●		●	
It is possible to discover the internal working of the program or learn about the implementation of special features or algorithms, coding techniques, and sensitive information by exploiting the Java bytecode.			●	
Data validation schemes or data processing can be revealed by thoroughly exploiting the Java class file.	●		●	
Bytecode can systematically be instrumented to introduce new logic for further exploitations. Thus, some internal functionality or values of local variable's can be altered during the attack (patching).	●			

Table 3.2: Threat Classifications for the Java Bytecode Vulnerabilities

CHAPTER 4 VULNERABILITY ASSESSMENT – A CASE STUDY

4.1 Overview

Using the Java bytecode vulnerability taxonomy developed in the previous section, the next step is to perform a vulnerability assessment on a real-life Java application. The purpose of this exercise is to assess the subject application by performing an intellectual penetration on the compiled bytecode contained in the Java class files. This vulnerability assessment can be classified as intellectual penetration because its ultimate objective is to demonstrate how the application's functionality can be exposed without having access to the source code.

Neither the subject application is installed nor has the program's documentation been referred to before performing this study. The reason for this is that the hacker may not always have access to the final installable version of the appplication or the complete set of class files, so that he or she may not be able to execute the application before exploiting it. However, a malicious user could still try in order to exploit the available subset of class files to retrieve any important information, or merely to learn more about the application as a basis for further attacks.

The complete process of vulnerability assessment is depicted in Figure 4.1

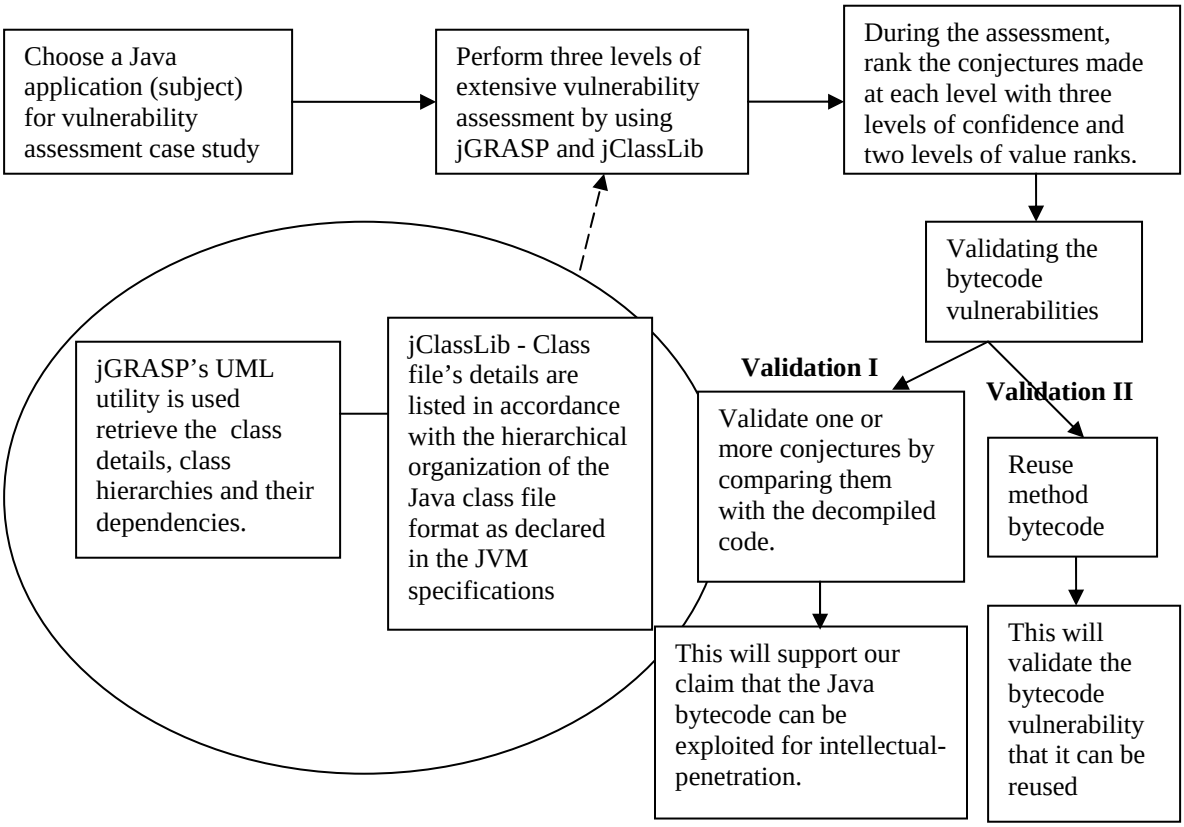

Figure. 4.1 Process depicting vulnerability assessment.

4.2 Vulnerability Assessment Approach

The following tools will be used during the process of vulnerability assessment and to exploit the bytecode.

4.2.1 Tool One: jGRASP

Purpose:

- To develop UML class diagrams for each package.
- To retrieve class details such as the names, signatures, and data types of the class elements.

- To visualize and understand the class hierarchy and class dependencies.

Expected Results:

- Taxonomy Class I: Names of the packages, classes, interfaces, methods, and fields are the intellectual assets of any application. Knowledge about these details can be exploited in many ways, as explained during the taxonomy development.
- Taxonomy Class II: A hacker can exploit details such as the methods' signatures, data types of the fields, etc.
- Taxonomy Class III: Class diagrams, class hierarchies and dependencies can lead to component-penetration attacks.
- Taxonomy Class IV: The above information can provide some insights about the functionalities and the overall architecture of the classes, packages, and the application as a whole.

4.2.2 Tool Two: jClassLib

Purpose:

- To view the Java class files details at bytecode level.
- To list the class file's details in accordance with the hierarchical organization of the Java class file format, as declared in the JVM specifications.
- To view arrays of constant pools, bytecode details for each method, fields, and attributes, all in assembly code format.

Expected Results:

- Get to know more about the code of each method at the bytecode level. These details can provide a deeper perception of the method's functionality.
- One can learn details such as the data types, the values of internal constants, and which other class methods or Java class libraries are being used by the current method to complete its operation.
- Thus, the functionality and the architecture of each class and package can be deduced more completely.

4.2.3 Vulnerability Assessment Strategy

A bottom-up approach was taken for this vulnerability assessment, which was performed at three levels.

Level 0: Component/Class/Interface level

Level 1: Package Level

Level 2: Application level

Level 0: During the 'level 0' assessment, a conjecture about the working of each component was made by exploiting each field and the methods declared in that class. jGRASP and jClassLib were used during this vulnerability assessment at the component level.

The Level 0 attack was carried out as follows**:**

- Study each *component* and try to learn more about its functionality.
  - Conjecture the purpose of each *field*.
  - Conjecture the function of each *method*.

Level 1: Speculations about the overall working of the package was done at 'level 1'. This assessment was based upon the conclusions drawn during the 'level 0' assessment, UML class diagrams, class hierarchies and the class dependencies found in that package.

The Level 1 attack was carried out as follows**:**

- Study each *package* and try to learn more about it.

- Conjecture the purpose and function each *Component* of that package.

Level 2: Exploiting each of the class files and the detailed study of each component led to a set of assumptions about the overall architecture and the functionality of the application as a whole.

The Level 2 attack was carried out as follows**:**

- Learn about the application as whole without having access to the source code

- Conjecture the purpose and function of each package.

4.2.4 Confidence Ranking

The conclusion drawn for each class, its elements, and the package was assessed using a three-level ranking. This ranking was based on the confidence of the assumptions made about the component's functionality. The three levels of confidence ranking were: low, medium, and high. Once the vulnerability assessment was completed, then any hacker would be most interested in further exploiting the components that have a 'high' confidence level because there is a better chance that the hacker can gain more information by exploiting them. The accuracy of the vulnerability assessment is based on the several factors, as stated below:

- **Size:** When the byte code length is comparatively small, it is easier to assess. The

longer is the bytecode, the chances of getting the bytecode assessment correct are less.

- **Dependencies:** If a component or method has fewer dependencies, the assessment can be done more accurately.
- **Method declarations**: The lower the numbers of methods defined in a class, the better the conjectures that can be made about that class's purpose and functionality.
- **Complexity**: If the bytecode of any method has many conditions and loops, its assessment becomes more difficult.
- **Literals**: If string constants are found in the bytecode, the assessment becomes easier.
- **Errors**: Any error messages found inside the bytecode give more clues to that method's functionality.

If it is difficult to draw any apparent conclusions and the assessment is based on mere guesswork, the ranking will be 'Low', while conjectures made on concrete evidence get a 'High' ranking. Conclusions that fall somewhat between these two extremes, receives a 'Medium' ranking.

4.2.5 Value Ranking – How reusable the component or the method is.

The bytecode of any Java application can be exploited and its components or methods can be reused. During the vulnerability assessment, each method and class is given a 'Value Ranking'. This ranking represents how important or useful it is to reuse that class or the method. Each class and method are assigned two levels of 'Value Ranking', either 'Low' and 'High'.

Any component's reusable value depends upon the following characteristics:

- **Complexity**: If the class is highly dependent on other components, then it is difficult to reuse, since it is necessary to track all its dependencies and related entities in order to reuse that class. This may increase the complexity.
- **Abstractness**: Some methods support the functionality of other methods. Generally such methods have no reusable value.
- **Importance**: The methods and classes that contain important calculations, critical algorithms, functions that require a lot of effort to develop from scratch, etc. are good candidates for reuse. Any hacker would prefer to reuse them rather than writing them from scratch.

The 'High' and 'Low' rankings assigned to the classes and their methods was based on the following criteria:

- The methods that have mathematical operations would be good candidates for reuse, since anyone who is developing a calculator application is likely to prefer to reuse these methods rather than developing them from scratch.
- The methods that perform graphing operations would be good candidates for reuse and are thus assigned 'High' rank, since these operations can be reused by any application that implements graphs.
- In order to reuse any method, its bytecode assessment must provide sufficient insights into its functionality and purpose. It would be difficult to reuse any method whose bytecode assessment has a lower degree of confidence. Methods whose functionality is not properly understood will not be good candidates, so they have been assigned a 'Low' rank.

4.2.6 Results

The ‘Open Calcualtor’ application was chosen for this case study. It is a Java based opensource application with a GNU General Public Licensed (GPL). The installable jar archive of this program was obtained from SourgeForge (http://www.sourceforge.net). There were three main resons for chosing this application. First, it is a J2SE application and thus was the proper candidate for this research. Second, this application might have some functionalities performing important arithmetic and graphing operations. Thus, it might be useful to assess vulnerabilities of the components controlling such functionalities. Third, since SourceForge provides applications with the public license and these are free to use, this application was a good candidate to avoid any legal troubles.

Table 4.1 summarizes the results of the vulnerability assessment of the ‘Open Calculator’application. Appendix C gives the detail results of the vulnerability assessment.

Class	**Conjecture**	**Confidence Rank**	**Value Ranking**
Exec	This creates another process to execute some system command.	High	Low
C	It is difficult to make any conjecture about this class’s responsibility. Its bytecode had some arithmetic operations and a string append function.	Low	Low
compare	This class seems to provide string comparison functions.	Medium	Low
procentOf	This class has a method that calculates percentage.	High	Low
Rand	Its bytecode has a ‘sine’ calculating function. It is difficult to demine its exact purpose.	Medium	Low
random	Random generation operation.	High	Low
sumIntegral	This class seems to perform various arithmetic operations, but it is difficult to make conjecture about its true functionality.	Low	Low

timeMs	Returns current time in milliseconds	High	Low
funcRunner	This class includes functions necessary for finding and reading a class file, retrieving its parameters and types, and running the function declared in that class file. The ultimate purpose of this class seems to run the function declared in the 'opencalculator.api.func' package.	Medium	High
ioAble	This seems to include the function definitions necessary to handle the input-output for this application.	High	Low
OCError	This abstract class has a variable defined to handle the various types of errors possible while running the Open Calculator application.	Medium	Low
OCPError	This returns the type and the line number where the error has occurred. This error seems to occur while running some types of program.	High	Low
OCprogramError	This returns the line number and the actual error that ccurred during the execution of some type of program.	High	Low
OCSyntaxError	Its purpose is to encapsulate functionalities related to syntax errors. Based on the assessment, one can make conjecture that these syntax errors might be occurring when 'Open Calculator' runs the functions or the programs.	Medium	Low
OCVariableError	This class encapsulates functionalities to represents the variable errors which are a type of Open Calculator errors. But, at this stage it is difficult to judge that what types of variable error may occur in this application.	Medium	Low
OpenCalculate	This class has methods responsible for performing arithmetical, trigonometric, and logical operations.	Medium	High
OpenCalculateKomando	One can conclude this class has functionalities that perform the operations on some functions, program, variable, strings etc. All these functions have string operations, error messages, and the objects instances of the 'OCSynatxError' class. It seems this Open Calculator application might be accepting user commands from the command prompt, since this class seems to have many string operations.	Medium	Low

OpenCalculateSatts	The member methods such as HittaTal()', 'HittaVariabel()'[sic],'HittaOperator()', 'HittaFunktion()', 'HittaParentes()' perform operation on some numbers, variables, operators, functions, and parentheses respectively. It seems that these are found in the user command that might have been entered at the command prompt interface. The other methods of this class seem to perform operations on the variables enclosed in the parentheses.	Medium	Low
OpenCalculateSatts$1	Nothing can be concluded about its purpose and functionality since no information is found for this class.	Low	Low
OpenCalculateSatt$ Parentes	This is the inner class of the 'OpenCalculateSatt', as its name suggests, but its assessment does not provide enough information to conclude anything about its purpose and functionality. All the methods declared in this class access the fields 'opencalculator.api.OpenCalculatorSatts$Parentes. parenteser'and 'opencalculator.api.OpenCalculatorSatts$Parentes. antal'. Nothing more can be concluded except this class seems to perform operations on the variable enclosed in the parentheses.	Low	Low
program	The bytecode of this method has only one significant instruction - 'opencalculator.api.programReturn<init>>'. Nothing can be concluded about its functionality except, that it is dependent on the class 'programReturn'.	Low	Low
programKomando	This is an abstract class and it provides a method definition that returns some program type.	High	Low
programList	This class has methods which operate on some 'programs', its variables, code, and operators. The string manipulation functions suggest that this class is operating on some type of code. The presence of some of the methods implies that this class controls some types of commands. This application has different components to control 'functions' and 'programs'. Thus, one can assume that the 'functions' and 'programs' have different responsibilities for this application. The class 'programList' has responsibilities for controling some 'programs' for this application.	Medium	High

Table 4.1: Summary of Vulnerability Assessment.

4.2.7 Validations

The final step of this vulnerability assessment was to validate our hypothesis that the Java bytecode is vulnerable and could be exploited to accomplish intellectual penetration.

Two types of validations were performed as follows:

4.2.7.1 Validation I:

4.2.7.1.1 Purpose**:** The purpose of this validation is to exhibit that the conjectures made during the case study are accurate. In other words, the purpose is to validate that one can reveal the functionality of any Java application and its components by exploiting the class files. validate

Four methods of various confidence levels were chosen and their class files were reverse engineered using a Java decompiling tool 'Jad'. The methods' bytecode, decompiled code, and the bytecode assessment results are documented in Appendix E.

1) Method Name: runFunc()
   Class Name: funcRunner
   Confidence Rank: Medium
2) Method Name: findClass()
   Class Name: funcRunner
   Confidence Rank: Medium
3) Method Name: FUNCtimeMs()
   Class Name: timeMs
   Confidence Rank: High

4) Method Name: countUtanParantes()

   Class Name: OpenCalculate

   Confidence Rank: Low

4.2.7.1.2 Results and Discussion: The following are the results of the comparisons made between the decompiled code and the bytecode assessments for all four methods. Each method's bytecode, decompiled code, and the assessment results are documented in Appendix D.

1) Method Name: runFunc()

- The conjecture that string manipulation functions are operating on the parameter passed to it is correct. The name of the string argument is's' and the functions 'substring()' and 'indexOf()' operates on this string argument.
- The assumption about the 'append ()' function is not completely correct. The strings "opencalculate.api.func." and "FUNC" are being appended but the method 'findClass()' is invoked for the string which is created by appending the string "opencalculate.api.func." and then the method 'getMethod()' is executed on the string created by appending "FUNC". Thus, the sequence of method invocation is difficult to judge by looking at the bytecode, unless the bytecode is assessed dynamically.
- The conclusion that the methods 'funcRunner.getParameters()' and 'funcRunner.getTypes()' operate on a class file or a class object is wrong. It actually operates on the string argument, not on the class or the class file, lthough

it seems to be true conclusion that these methods are accessing the parameters and their types.

- The assumption made about the purpose of the method call ‘getMethod()’ is correct; it does seem to invoke the method with the word ‘FUNC’ in it.

2) Method Name: findClass()

- The ‘replace()’ functions replaces the character “.” with “/”. The assessment indicated that this function was being called, but it was not possible to determine which character was being replaced with which character.
- The function ‘append()’ appends the string “.class” to the string argument.
- The assumption made about the purpose of method ‘FileInputStream.available()’, ‘FileInputStream.read()’, and the instruction ‘anewarray’ are correct.
- The assumption that the ‘if…equals’ conditions are comparing some strings is incorrect. These conditions are actually checking that if the object of type ‘FileInputStream’ is not null, then this ‘FileInputStream’ is closed and the exception is caught.

3) Method Name: FUNCtimeMs()

- The conjecture made about this method’s purpose is correct. It returns the current time in milliseconds.

4) Method Name: countUtanParantes()

- The long length of the bytecode made this bytecode assessment difficult.
- The observations made about the methods ‘opencalculator.api.OpenCalculate.HittaMinustal()’, ‘opencalculator.api.OpenCalculate.UtforFunktioner (), and

‘opencalculator.api.OpenCalculate.UtforOperator()’ were correct, but the purpose of these method calls was not revealed during the bytecode assessment.

- The purpose of the method calls ‘opencalculator.api.OpenCalculateSatts.size()’ and ‘opencalculator.api.OpenCalculateSatts.getValue()’ were not anticipated in the assessment, but a review of the decompiled code provided more insights about their purpose. If the value returned by the function ‘OpenCalculateSatts.size()’ is more then 1, then the error message is displayed and an object is instantiated of type ‘OCSyntaxError’.
- The bytecode assessment for this method was difficult since it was calling many external methods.

4.2.7.2 Validation II:

4.2.7.2.1 Purpose**:** The purpose of this validation was to demonstrate that a component can be reused. The following three methods were selected on the basis of their value ranks and their bytecode is reused in another Java application:

1) Method Name: getFunctionValue()
   Class Name: OpenCalculate
   Value Ranking: High
2) Method Name: getOperationValue()
   Class Name: OpenCalculate
   Value Ranking: High

3) Method Name: FUNCprocentOf()

Class Name: procentOf

Value Ranking: High

4.2.7.2.2 Results and Discussion**:** There are many other components and methods available in the ‘Open Calculate’ application with ‘High’ value ranks that show potential for reuse. However, their reuse value depends on the functionality and the purpose of the application which is reusing them. Some methods are better candidates for reuse than others for particular applications.

Appendix E has the details of the application which is reuses these methods. The following are some of the challenges faced during this validation.

Challenge 1: The methods ‘getOperationValue()’ and ‘getFunctionValue()’ were declared private, so they can only be called withine the scope of the class ‘OpenCalculate’.

Resolution: It was necessary to change their access modifier in order to reuse them. A ‘HEX’ editor was used to modify the class files. The review of this ‘HEX’ editor is given in Appendix G.

In order to locate and modify the particular bytecode which is responsible for the access modifiers, the Java class file vulnerability was exploited. The class file format is very systematic and adheres to the JVM specification. It is described by a series of data structures that represent the class file itself, its methods, its fields, and its attributes. The access flag for the ‘private’ modifier is a two byte integer with value ‘0x0001’ and the ‘Flag Name’ is ‘ACC_PUBLIC’. The bytecode ‘0x0001’

required to be replaced with ‘0x0002’ which is the access_flag for the ‘public’ modifier. The offset values of the access_flag byte code for the methods ‘getOperationValue()’and ‘getFunctionValue()’ are Offset ‘3943 = 0xf67’ and ‘3165=0xc5d’ respectively. The responsible byte was replaced with the byte responsible for the ‘private’ access_flag.

Challenge: In order to reuse these methods, it was necessary to extract their dependencies.

Resolution: The source files of the application that is reusing these methods were included in the directory of the ‘Open Calculator’ application and the package name was added at the beginning of each source file.

CHAPTER 5 CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

This study affirms the underlying vulnerabilities of the Java bytecode. It is evident that the well-defined format of the Java class files makes it feasible to exploit them even without installing or running the application. The focus of this research was to conduct a vulnerability assessment of Java bytecode in order to reveal its vulnerabilities.

Java bytecode exploitations can pose threats to the authenticity, integrity, confidentiality, and the availability of the Java programs. The bytecode vulnerability taxonomy described in Chapter 3 was developed on the basis of the above classifications and can be used to increase our overall understanding of the bytecode vulnerabilities. It was applied during the vulnerability assessment of a real-life Java application.

An intellectual penetration was performed on a Java application during the case study in Chapter 4. The intention of this extensive vulnerability assessment was to learn as much as possible about the application and its components. Four methods from the subject application were reused in another Java program, which demonstarted that the component penetration can be carried out by exploiting the Java bytecode. The vulnerability assessment results and their validations confirmed that the Java bytecode can be exploited to carry out intellectual and component penetrations.

The following are some of the highlights of the validations:

- The similarities found in the conjectures made from the bytecode assessment and the decompiled code confirms that the Java class file can be exploited to reveal secrets of the program, its functionality, purpose, and the overall architecture.
- The accuracy of these conjectures depends on various factors, such as the quality of the reverse engineering tools, and the knowledge and abilities of the person who is performing the assessment.
- The following details are comparatively tricky to infer by statically assessing the bytecode:
    - The precise sequence of method invocations.
    - Specific details about the loops and the conditional branches.
    - Values of the variables with the ‘final’ modifier.
    - During the assessment of any particular method, it is difficult to keep track of the external method calls which are made for this method.
- The use of a static bytecode assessment offers some advantages over the traditional decompilation tools. It is a better choice when the hacker wants to determine the class details, their functionalities, and gain a high level understanding of the application and its components with less effort and in a shorter time without the need to examine every detail and the complexities of the decompiled code.
- The demonstration of method reuse by a Java program exposed the bytecode vulnerability that it can be extracted and reused by a hacker.

Securing Java Bytecode: It is virtually impossible to develop a software application that confers absolute protection and which can never be hacked. However, one can make it

difficult to exploit the Java bytecode by making it more difficult to crack. The following are some common practices that can be used to make bytecode harder to exploit:

- Obfuscation: Obfuscation is a technique that can be used to scramble the class file so that it becomes harder to understand the decompiled code. There many tools available for obfuscation.
- Java Cryptography API [Java API 2002]: The Java Cryptography API provides libraries that can be used to encrypt and decrypt the code, protecting the integrity of the data with a message digest, and incorporating other techniques that can be used to protect the core files from hacking and patching.
- Bytecode Hosing: The systematic pattern or the structure of the class file makes it more vulnerable. Bytecode hosing is a technique that breaks these recognizable patterns by adding fake instruction sequences.
- Tamper Proofing [Collberg and Thomborson 2002]: Tamper proofing techniques disable some or all of the program functionalities once unwanted modifications of the class files have been detected.

5.2 Future Work

Since promising results were found while performing the vulnerability assessment on the Java application, further bytecode assessment could be very useful for both developers and the information security community. In order to reveal all the possible bytecode vulnerabilities, the following work needs to be pursued:

- Elaborate the Taxonomy of Bytecode Vulnerabilities:

A more detailed review of the taxonomies developed for various software vulnerabilities is required. An intense study of past research will help classify the bytecode vulnerabilities into more specific categories. Further study on software vulnerability is also needed in order to help identify the characteristics and the consequences of possible bytecode exploitations.

- Complete the vulnerability sssessment of other packages of the 'Open Calculator' application:

Further assessment of the 'Interface' package of this application needs to be completed. This additional assessment would give a more complete picture of the detailed functionalities of each component of this application. The confidence level of the assessment results could rise with the help of some additional tools which can help to accomplish a thorough bytecode assessment.

- Develop a tool to carry out a component penetration attack on the 'Open Calculator' application:

In order to establish other aspects of the bytecode vulnerability, it is necessary to carry out a component penetration attack on the 'Open Calculator' program. A tool needs to be designed and developed to demonstrate this vulnerability. The tool should be able to accomplish the following:

- To import all the classes declared in the subject application.
- To parse each class file into arrays of structures as defined in the JVM specification.

- To decompose and display the entire application into separate and manageable components.
- When the user selects a class or a method for potential reuse, the tool should be able to extract all the dependencies required in order to reuse that class or method.

- Perform intrusion penetration attack:

Carry out the intrusion penetration attack on any other Java application that performs user authentication. The underlying bytecode needs to be exploited in order to break the authentication process and gain unauthorized access. Demonstrating this type of attack will reveal the other aspects of the Java bytecode vulnerabilities.

REFERENCES

ABBOTT, R. P. and DONNELLEY, J. E. 1976. *Security Analysis and Enhancement of Computer Operating Systems, National Bureau of Standard Report NBSIR TR No. 76-1041*. ICST, Gaithersburg, Md.

BISHOP, M. 1999. Vulnerability Analysis. In *Proceedings of the Second International Symposium on Recent Advances in Intrusion Detection*, September 1999, 125-136.

CHESWOCK, W. R. and BELLOVIN, S. M. 1994. *Firewall and Internet Security: Repelling the Wily Hacke.* Addison-Wesley Publication Company, Reading, MA.

COLLBERG, C, and THOMBORSON, C. 2002. Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection. *IEEE Transactions on Software Engineering, 28*, 8, 735-746.

HAGGAR, P. 2001. Understanding bytecode makes you a better programmer, *IBM resource for developers DeveloperWorks*. http://www-128.ibm.com/developerworks/ibm/library/it-haggar_bytecode/.

HOWARD, J. D. and LONGSTAFF, T. A. 1998. *A Common Language for Computer Security Incidents, Sandia Technical Report TR No. SAND98-8667.* Sandia National Laboratories. http://www.cert.org/research/taxonomy_988667.pdf.

JAD. 1997. Java Decompiler. http://kpdus.tripod.com/jad.html.

JAVA API. 2002. Java Cryptography Architecture: API Specifications & References. http://java.sun.com/j2se/1.4.2/docs/guide/security/CryptoSpec.html.

JODE. 1998. Java Decompiler, release 1.1.1. http://jode.sourceforge.net.

KALINOVSKY, A. 2004. *Covert Java: Techniques for Decompiling, Patching, and Reverse Engineering*. SAMS Publications, Indianapolis, Indiana.

LINDHOLM, T. and YELLIN, F. 1999. The Java Virtual Machine Specification: Second Edition. *Sun Microsystems Inc*. http://java.sun.com/docs/books/vmspec/2nd-edition/html/ VMSpecTOC.doc.html

MOCHA. 1996. Java Decompiler. http://www.brouhaha.com/~eric/ software /mocha.

POWER, R. 1996. Current And Future Danger: A CSI Primer of Computer Crime & Information Warfare. *CSI Bulletin.*

SIMPSON, G. G. 1995. *The Principles of Classification and a Classification of Mammals.* New York.

SUN MICROSYSTEMS INC. 1996. A Sun Developer Netwrok Site. http:// www. Java.sun.com/

SURYADEVARA, V. and AHMED, A. 2004. Security Vulnerabilities –Reverse Engineering, Project Report, ECE 478/578.

UMPHRESS, D. 2004. Software as an exploitable source of intelligence. *The College of Aerospace Doctrine, Research and Education (CADRE) Quick-Looks.*

VENNERS, B. 1999. Bytecode basics: A first look at the bytecode of the Java Virtual Machine. *JavaWorld*. http://www.artima.com/underthehood/bytecode.html.

WEBOL 1998. Merriam-Webster OnLine: WWWebster Dictionary. http://www.m-w.com/dictionary.htm

APPENDICES

APPENDIX A

1. API Package – UML Class diagram 1

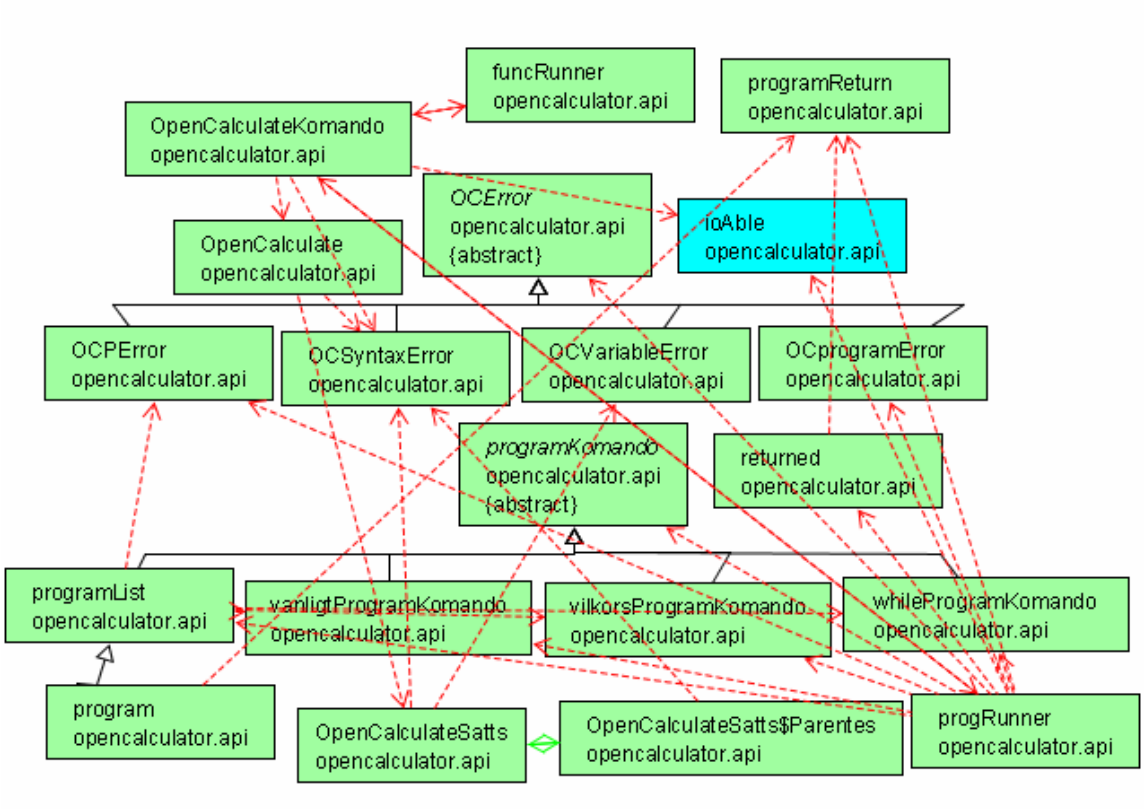

Figure.1: UML Class Diagram 1 – API Package

2. API.func package - UML class diagram 2

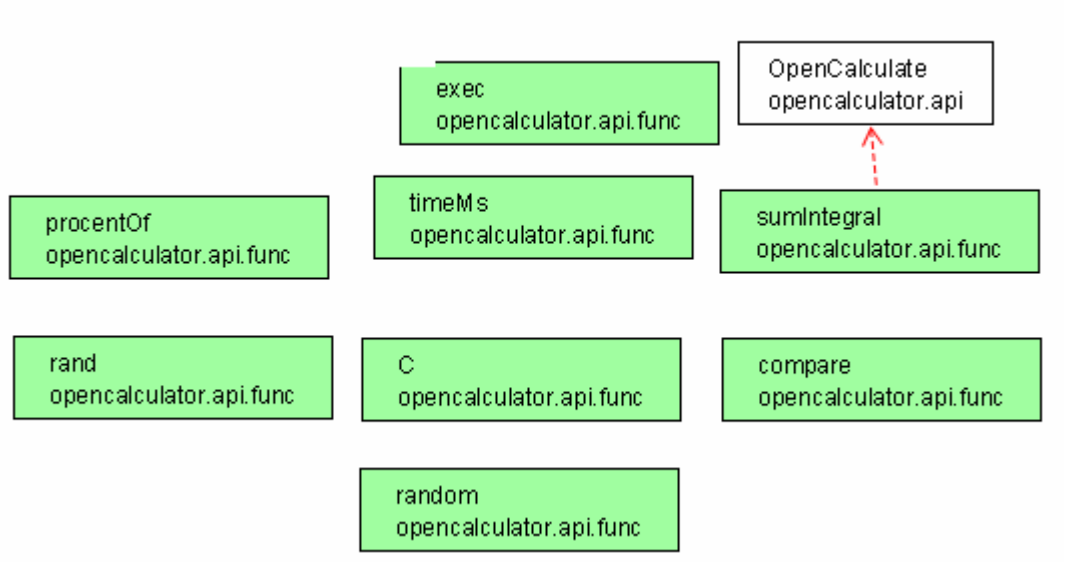

Figure.2: UML Class Diagram 2 – API.func Package

3. Interface package (a) - UML class diagram 3

Figure.3: UML Class Diagram 3 – Interface Package (a)

4. Interface package (b) - UML class diagram 4

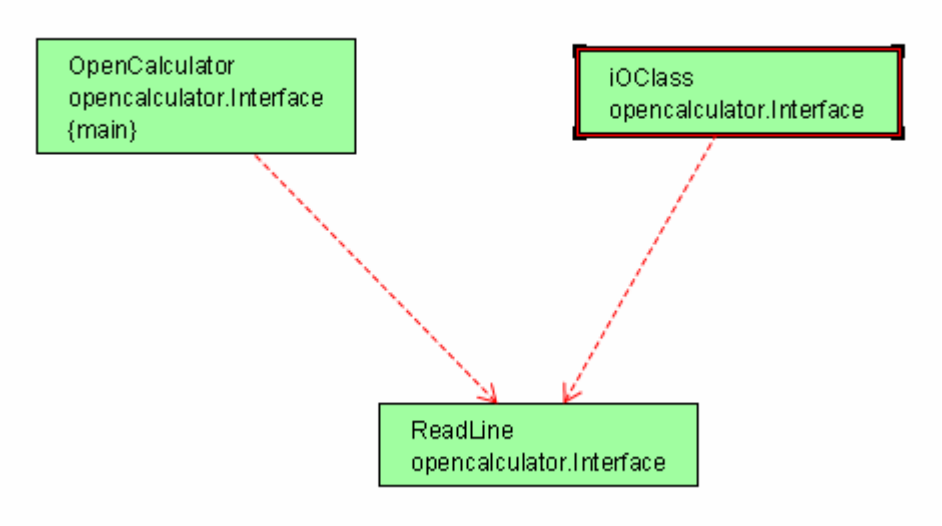

Figure.4: UML Class Diagram 4 – Interface Package (b)

APPENDIX B

API Package

Class Name**:**	OCError		
Fields:		typ	int
Methods:	public	getTypeOfError()	int

Class Name:	OCPError		
Fields:		Line	int
		typOfProgramError	int
Methods:	public	getType()	int

Class Name:	OCSyntaxError		
Fields:		typOfSyntaxError	int
Methods:	public	getType()	int

Class Name:	OCVariableError		
Fields:		variabel[sic]	int
Methods:	public	getVariable()	int

Class Name:	OCprogramError		
Fields:		line	int
		error	opencalculator.api.OCError
Methods:	public	getError()	opencalculator.api.OCError
	public	getLine()	int

Class Name: OpenCalculateSatts$Parentes

Fields:	private	antal	int
	private	parenteser	boolean[]
	final	this$0	opencalculator.api.OpenCalcualteSatts
Methods:	public	addHoger()	void
	public	addVanster()	void
	public	deleteInersta()	void
	public	getAntal()	int
	public	getInersta()	int

Class Name: OpenCalculate

Fields:		variables	java.lang.Objects[]
	private	angle	int

66

Methods:	private	HittaMinustal(opencalculator.api.OpenCalcualteSatts)	opencalculator.api.OpenCalcualteSatts
	private	UtforFunktioner(opencalculator.api.OpenCalcualteSatts)	opencalculator.api.OpenCalcualteSatts
	private	UtforOperator(opencalculator.api.OpenCalcualteSatts,int,int)	opencalculator.api.OpenCalcualteSatts
	public	count(string)	double
	private	countUtanParantes(opencalculator.api.OpenCalcualteSatts)	java.lang.Double
	private	getFunctionValue(int,double)	double
	private	getOperationValue(int,double,double)	double
	private	getValue(opencalculator.api.OpenCalcualteSatts)	double
	public	getVariable(int)	double

public getvariable(int,java.lang.Double) void

Class Name: progRunner
Fields: private io opencalculator.api.ioAble
komandoTolk opencalculator.api.OpenCalcualtorKomando
variables java.lang.Object[]

Methods: private checkDoProgramKomando(opencalculator.api.vanligtProgramKomando)
boolean
public checkDoReturnKomando(opencalculator.api.vanligtProgramKomando)
void private
doProgramKomando(opencalculator.api.programKomando)
void private
doVanligtProgramKomando(opencalculator.api.vanligtProgramKomando)
void private
doVilkorsProgramKomando(opencalculator.api.vilkorsProgramKomando)
void private
doWhileProgramKomando(opencalculator.api.whileProgramKomando)
void
private findVariable(string) int
private getBorgan(string,string) string
public run(java.io.File) java.lang.Object
private runList(opencalcualtor.api.programList) opencalcualtor.api.programReturn

Class Name: returned
Fields: R opencalculator.api.programReturn
Methods: getreturn() opencalculator.api.programReturn

Class Name: OpenCalculateSatts
Fields: private Elementen java.util.Vector

		Parenteser	opencalculator.api.OpenCalculateSatts$Parentes
	private	vairabler[sic]	java.lang.Object[]
Methods:	private	HittaFunktion(int,string)	int
	private	HittaOperator(int,string)	int
	private	HittaParentes(int,string)	int
	private	HittaTal(int,string)	int
	private	HittaVariabel(int,string)	int
	public	dellnersteParentes(java.lang.Double)	void
	public	getIdentifier(int)	int
	public	getInersteParentes()	opencalculator.api.OpenCalculateSatts
public		getValue(int)	double
	public	getVariable(int)	double
	public	remove(int)	void
	public	setNumber(int,double)	void
	public	setVariable(int,java.lang.Double)	void
	public	size()	int

Class Name: OpenCalculateKomando

Fields:		AttSkicka	java.lang.Object
		Variables	java.lang.Object[]
		io	opencalculator.api.ioAble
Methods:		DoFunc(string)	boolean
		DoProg(string)	boolean
		DoString(string)	boolean
		DoTilldelning(string)	boolean
		DoUtryck(string)	boolean
		DoVariable(string)	boolean
	private	checkIfvalid(string,int)	boolean
	public	run(string)	java.lang.Object

Class Name: funcRunner

Fields:		Vairables[sic]	java.lang.Object[]
Methods:	protected	findClass(string)	java.lang.Class
	private	getparameters(string)	java.lang.Object[]
	private	getTypes(java.lang.Object[])	java.lang.Class[]
	public	runFunc(string)	java.lang.Object

Class Name:	programKomando		
Fields:	private	type	int
Methods:	public	getType()	int

Class Name:	whileProgramKomando		
Fields:	private	Komando	opencalculator.api.programKomando
	private	Vilkor	string
	private	line	int
	private	whileOrDo	boolean
Methods:		getKomando()	opencalcualtor.api.programKomando
		getLine()	int
		getVilkor()	string
	public	setKomando(opencalculator.api.programKomando)	void
		setLine(int)	void
		whileOrDo()	boolean

Class Name:	vilkorProgramKomando		
Fields:	private	Komando	opencalculator.api.programKomando
	private	Vilkor	string
	private	elseKomando	opencalculator.api.programKomando
	private	line	int
	private	whitElse[sic]	boolean
Methods:	public	getElseKomondo()	opencalculator.api.programKomando

	public	getKomando()	opencalculator.api.programKomando
		getLine()	int
	public	getVikor()	string
	public	setElseKomando(opencalculator.api.programKomando)	void
public		setKomando(opencalculator.api.programKomando)	void
		setLine(int)	void
	public	whitElse()[sic]	boolean

Class Name: vanligtProgramKomando

Fields:		Komando	string
		line	int
Methods:		getKomando()	string
	public	getLine()	int
	public	setLine(int)	void

Class Name: programList

Fields:	private	forstaLines	boolean
		komandoList	java.util.Vector
		nSetLine	int
	private	nextCounter	int
	private	radnummer	java.util.Vector
Methods:	private	checkLast(opencalculator.api.programKomando)	int
	public	clearCounter()	void
	private	doTokenizon(string)	java.util.Vector
	private	getKomandoList(java.util.Vector)	java.util.Vector
	private	getKomandoStrings(java.io.LineNumberReader,int)	java.util.Vector
public		getNext()	opencalculator.api.programKomando

private makeKomando(java.util.Vector,int[]) opencalculator.api.programKomando
private makeVilkorsKomando(java.util.Vector,int[]) opencalculator.api.vilkorsprogramKomando
makeWhileKomando(java.util.Vector,int,int[])
opencalculator.api.whileprogramKomando private
setLineNumber(java.util.Vector) java.util.Vector
private setLineNumberThis(opencalculator.api.programKomando)
opencalculator.api.programKomando
public size() int
private vilkorsKomando(string) boolean
private whileKomando(string) int

Class Name: program
Fields: NONE
Methods: public run() opencalculator.api.programReturn

Class Name: programReturn
Fields: retrunElement java.lang.Object
returnType int
Methods: java.lang.Object getObject()
getReturnType() int

Interface Name: ioAble
Fields: NONE
Methods: public abstract get() string
public abstract print(java.lang.Object) void
public abstract println(java.lang.Object) void

API.FUNC package

Class Name: exec
Fields: NONE
Methods: public static FUNCexec(string,java.lang.Object[]) java.lang.Object

Class Name: procentOf
Fields: NONE
Methods: public static FUNCprocentOf(java.lang.Double.java.lang.Double,java.lang.Object[])
java.lang.Object

Class Name: timeMs
Fields: NONE
Methods: public static FUNCtimeMs(java.lang.Object[]) java.lang.Object

Class Name: sumIntegral
Fields: NONE
Methods: public static FUNCsumIntegral(string,string,java.lang.Double,java.lang.Double,java.lang.Object[])
java.lang.Object

Class Name: rand
Fields: NONE
Methods: public static FUNCrand(java.lang.Object[]) java.lang.Object

Class Name: C
Fields: NONE
Methods: public static FUNCC(java.lang.Double.java.lang.Double,java.lang.Object[])
java.lang.Object

Class Name: compare
Fields: NONE

Methods: public static FUNCcompare(string,string,java.lang.Object[]) java.lang.Object

Class Name: random
Fields: NONE
Methods: public static FUNCrandom(java.lang.Object[]) java.lang.Object

Interface Package

Class Name: OpenCalculator
Fields: NONE
Methods: public static main(string[]) void
Public static printErrInformation(olpencalculator.api.OCError) void

Class Name: ReadLine
Fields: NONE
Merthods: public static getString() string

Class Name: IOClass
Fields: NONE
Methods: public get() string
public print(java.lang.Object) void
public println(java.lang.Object) void

Class Name: getNameForm$2
Fields: NONE
Methods: public actionPeformed(java.awt.event.ActionEvent) void

Class Name: getNameForm$1
Fields: NONE

Methods: public windowClosing(java.awt.WindowEvent) void

Class Name: getNameForm
Fields: ab[sic] opencalculator.Interface.program
private jButton1 javax.swing.JButton
private jLabel1[sic] javax.swing.JLabel
private jTextField1 javax.swing.JTextField
Methods: private exitForm(java.awt.WindowEvent) void
getFileName(opencalculator.Interface.WindowEvent) void
private initComponents() void
private jButton1ActionPerformed(java.awt.event.aCTIONeVENT) void

public static main(string[]) void

74 Class Name: graph$useVWAction
Fields: final this$0 opencalculator.Interface.graph
Methods: public actionPerformed(java.awt.event.ActionEvent) void

Class Name: graph
Fields: DeleteThisVW javax.swing.JButton
LoadThisVW javax.swing.JButton
TabbedPane javax.swing.JTabbedPane
Tabel[sic] javax.swing.JTable
ViewTableScroll javax.swing.JScrollPane
ViewWTable javax.swing.JTable
WiewWindow javax.swing.JPanel
addNewGraph javax.swing.JButton
deleteGraph javax.swing.JButton
drawGraphs javax.swing.JButton
gPainter opencalculator.Interface.graphPainter
graphFuncPanel javax.swing.JPanel

			graphPanel	javax.swing.JPanel
			saveThisVW	javax.swing.JButton
			scrollerForTable	javax.swing.JScrollPane
			storedVW	javax.swing.JTable
			storedVWScroll	javax.swing.JScrollPane
			useThisVW	javax.swing.JButton
Methods:	private		initComponents()	void

Class Name:	graph$3			
Fields:			canEdit	boolean[]
			Types	java.lang.Class[]
Methods:	public		getColumnClass(int)	java.lang.Class
	public		isCellEditable(int,int)	boolean

Class Name:	graph$2			
Fields:	final		this$0	opencalculator.Interface.graph
Methods:	public		getColumnClass(int)	java.lang.Class

Class Name:	graph$1			
Fields:			canEdit	boolean[]
	final		this$0	opencalculator.Interface.graph
			Types	java.lang.Class[]
Methods:	public		getColumnClass(int)	java.lang.Class
	public		isCellEditable(int,int)	boolean

Class Name:	program$3			
Fields:	final		this$0	opencalculator.Interface.proram
Methods:	public		actionPerformed(java.awt.event.ActionEvent)	void

Class Name:	graph$drawGraphAction			
Fields:	final		this$0	opencalculator.Interface.graph

Methods: public actionPerformed(java.awt.event.ActionEvent) void

Class Name:programEditor$3
Fields: NONE
Methods: public actionPerformed(java.awt.event.ActionEvent) void

Class Name: graphPainter
Fields: private Grapher java.util.Vector
private Names java.util.Vector
private Xmax double
private Xmin double
private Xscale double
private Ymax double
private Ymin double
private Yscale double
private raknare opencalculator.api.OpenCalculator
Methods: private initComponents() void
protected paintComponent() void
public setGraphs(java.util.Vector,java.util.Vector) void
public setViewWin(double,double,double,double,double,double) void

Class Name: programEditor$2
Fields: NONE
Methods: public actionPerformed(java.awt.event.ActionEvent) void

Class Name: graph$saveVWAction
Fields: final this$0 opencalculator.Interface.graph
Methods: public actionPerformed(java.awt.event.ActionEvent) void

Class Name: OCMainFrame$3

Fields:			canEdit	boolean[]
			types	Java.lang.Class[]
Methods:	public		getColumnClass(int)	java.lang.Class
	public		isCellEditable(int,int)	boolean

Class Name:	OCMainFrame$2			
Fields:			canEdit	boolean[]
			types	Java.lang.Class[]
Methods:	public		getColumnClass(int)q	java.lang.Class
	public		isCellEditable(int,int)	boolean

Class Name:	OCMainFrame$1			
Fields:	final		this$0	opencalculator.Interface.OCMainFrame
Methods:	public		windowClosing(java.awt.event.WindowEvent)	void

77

Class Name:	OCMainFrame			
Fields:	private		console1	opencalculator.Interface.console
	private		graph1	opencalculator.Interface.graph
	private		jTabbedPane1	javax.swing.JTabbedPane
	private		program1	opencalculator.Interface.program
Methods:	static		access$000(opencalculator.Interface.OCMainFrame,java.awt.event.WindowEvent)	void
	private		exitForm(java.awt.event.WindowEvent)	void
	private		initComponents()	void
	public	static	main(string[])	void

Class Name:	OCMain			
Fields:	private		HelpPane	calpa.html.CalHTMLPane
	private		console1	opencalculator.Interface.console
	private		graph1	opencalculator.Interface.graph
	private		jPanel1	javax.swing.JPanel

private	jScrollPane1	javax.swing.jScrollPane	
private	jTabbedPane1	javax.swing.JTabbedPane	
private	program1	opencalculator.Interface.program	
Methods: static	access$000(opencalculator.Interface.OCMain,,java.awt.event.WindowEvent)	void	
private	exitForm(java.awt.event.WindowEvent)	void	
	getTabbedPane()	javax.swing.JTabbedPane	
private	initComponents()	void	
public static	main(string[])	void	

Class Name:	OCMain$1			
Fields:	final	opencalculator.Interface.OCMain	this$0	
Methods: public	windowClosing(java.awt.event.WindoEvent)	void		
Class Name:	console$caretListener			
Fields:	final	this$0	opencalculator.Interface.console	
Methods: public	caretUpdate(javax.swing.event.CaretEvent)	void		

Class Name:	getNameForm$1		
Fields:	NONE		
Methods: public	windowClosing(java.awt.event.WindowEvent)	void	

Class Name:	getNameForm$2		
Fields:	NONE		
Methods: public	actionPerformed(java.awt.event.ActionEvent)	void	

Class Name:	console		
Fields:	private	EditorPane	javax.swing.JTextArea
private	History	java.util.Vector	
	HistoryPosition	int	
private	ScrollPaneConsole	javax.swing.JScrollPane	
	consoleFont	java.awt.Font	

priavte consoleMode boolean
private consolePanel javax.swing.JPanel
private fromKeyboard boolean
private geted[sic] boolean
private hej[sic] java.lang.Thread
private startOfKomand int
private stringGet java.lang.String
private thisThread java.lang.String
private tolk[sic] opencalculator.api.OpenCalculateKomando

Method: static access$1000(opencalculator.Interface.console) java.lang.Runnable

static access$300(opencalculator.Interface.console) boolean
static access$302(opencalculator.Interface.console,boolean) boolean

static access$400(opencalculator.Interface.console) javax.swing.JTextArea

static access$500(opencalculator.Interface.console) int
static access$502(opencalculator.Interface.console,int) int
static access$602(opencalculator.Interface.console,string) string
static access$702(opencalculator.Interface.console,boolean) boolean
static access$800(opencalculator.Interface.console) java.util.Vector
static access$900(opencalculator.Interface.console) java.lang.Thread
static access$902(opencalculator.Interface.console,java.lang.Thread) java.lang.Thread

public get() string
private initComponents() void

Class Name: OCMain2
Fields: private console1 opencalculator.Interface.console

	private	graph1	opencalculator.Interface.graph
	private	jEditorPane1	javax.swing.JEditorPane
	private	jScrollPane1	javax.swing.jScrollPane
	private	jTabbedPane1	javax.swing.jTabbedpane
	private	program1	opencalcualtor.Interface.program
Methods:	private	exitForm(java.awt.event.WindowEvent()	void
	private	initComponents()	void
	public static	main(string[])	void

Class Name: program

Fields:		Co[sic]	opencalculator.Interface.Co
		fileGetter	opencalculator.Interface.getNameForm
	private	jButton1	javax.swing.JButton
	private	jButton2	javax.swing.JButton
	private	jButton3	javax.swing.JButton
	private	jButton4	javax.swing.JButton
	private	jList1	javax.swing.JList
	private	jPanel1	javax.swing.JPanel
	private	jPanel2	javax.swing.JPanel
	private	jScrollPane1	javax.swing.JScrollPane
		korEttProgram	boolean
		main	oprencalculator.Interface.OCMain
Method	static	access$000(opencalculator.Interface.program,java.awt.event.ActionEvent)	void
	static	access$100(opencalculator.Interface.program,java.awt.event.ActionEvent)	void
	static	access$200(opencalculator.Interface.program,java.awt.event.ActionEvent)	void
	static	access$300(opencalculator.Interface.program,java.awt.event.ActionEvent)	

			void	
	createList()		void	
	createNew(string)		void	
private	initComponents()		void	
private	jButton1ActionPerformed(java.awt.event.ActionEvent)			void
private	jButton2ActionPerformed(java.awt.event.ActionEvent)			void
private	jButton3ActionPerformed(java.awt.event.ActionEvent)			void
private	jButton4ActionPerformed(java.awt.event.ActionEvent)			void

Class Name: programEditor

Fields:	private	console1	opencalculator.Interface.console
private	fil[sic]	java.io.File	
private	jButton1	javax.swing.JButton	
private	jButton2	javax.swing.JButton	
private	jPanel1	javax.swing.JPanel	
private	jScrollPane1	javax.swing.JScrollPane	
private	jTextPane1	javax.swing.JTextPane	

Methods:	private	exitForm(java.awt.event.WindowEvent)	void	
private	initComponents()	void		
private	jButton1ActionPerformed(java.awt.event.ActionEvent)			void
private	jButton2ActionPerformed(java.awt.event.ActionEvent)			void

Class Name: OCMain2$1

Fields:	NONE		
Methods:	public	windowClosing(java.awt.event.WindowEvent)	void

Class Name: program$1

Fields:	final	this$0	opencalculator.Interface.program
Methods:	public	actionPerformed(java.awt.event.ActionEvent)	void

Class Name: program$2

Fields: final this$0 opencalculator.Interface.program
Methods: public actionPerformed(java.awt.event.ActionEvent) void

Class Name: program$4
Fields: final this$0 opencalculator.Interface.program
Methods: public actionPerformed(java.awt.event.ActionEvent) void

Class Name: console$mouseListner
Fields: final this$0 opencalculator.Interface.program
Methods: public mouseDragged(java.awt.event.MouseEvent) void

Class Name: console$keyLyssna
Fields: final opencalculator.Interface.program this$0
Methods: public keyPressed(java.awt.event.KeyEvent) void

Class Name: programEditor$1
Fields: NONE
Methods: public windowClosing(java.awt.event.WindowEvent) void

APPENDIX C

I. Level 0 Assessment Results for the 'API.func' package:

a) Class Name: exec
Assumption: The class name 'exec' does not really imply about its functionality.

Total Fields: NONE

Total Methods: One
1. public static FUNCexec (java.lang.String, java.lang.Object [])
   returns: java.lang.Object

Assumption: The name and signature of the method 'exec' does not imply anything significant about its functionality.

Method Bytecode:
- 'FUNCexec()' is calling:
  - java.lang.Runtime.getRuntime()
  - java.lang.Runtime.exec(string) returns: java.lang.Process
- **Other information: '**FUNCexec ()' seems to return NULL for some condition.

Conjecture: According to the Java API documentation, 'Runtime.exec ()' accepts a specific system command as a string parameter and executes that string command in a separate process. It is very likely that the command names passed to the 'Runtime.exec()' are the same string parameters that are passed to the 'FUNCexec()' method. This method might be returning NULL in case the process is not created successfully.

Final Conclusion: Based on the above findings from bytecode exploitation of the 'exec' class, one can assume that the 'exec' might be creating another process to execute some system command.
Confidence Rank: High
Value Ranking: Low

b) Class Name: C
Assumption: The name 'C' of this class does not tell anything about its functionality.
Total Fields: NONE
Total Methods: One
1. public static FUNCC(java.lang.Double, java.lang.Double, java.lang.Object[])
   returns: java.lang.Object

Assumptions: The name and the signature of the method 'FUNCC()' does not tell anything about its functionality.

Method Bytecode:

- 'FUNCC()' is calling:
  - java.lang.Double.longValue() returns: long
  - java.lang.StringBuilder.append(string) returns: java.lang.StringBuilder
  - java.io.PrintStream.println(string) void

- **Other Information:**
  - Arithmetical operations: *Subtract division*, and *compare* for values of type long.
  - Returns reference to the java.lang.Object

Conjecture: The method 'FUNCC()' seems to perform some arithmetic operations on the two parameters of type double, which are converted into long. It is performing string appending operations.

Final Conclusion: It is difficult to make any conclusion about the purpose of this class. Since, this class is found under the 'func' package, one can assume that this class is performing some type of function for this Open Calculator application.

Confidence Rank: Low
Value Ranking: Low

c) Class Name: **Compare**

Assumption: Its name suggests that this class might be performing a comparison operation.

Total Fields: NONE
Total Methods: One

1. public static FUNCCompare(string, string, java.lang.Object[])
   returns: java.lang.Object

Assumption: From the method's name its signature, one can make an assumption that this method might be performing comparison between the two string parameters. The third parameter of type java.lang.Object[] does not tell us anything other about this method.

Method Bytecode:

- Function 'FUN compare()' is calling:
  - java.lang.String.equals(java.lang.Object) returns: boolean

- **Other Information**:
  - The 'string.equals()' method is called after two local variables are loaded into the stack for two references.
  - One 'if equals' condition found in the bytecode
  - Two instances of java.lang.Double are created and the instance initialization methods are called for them. This instance initialization method declaration for 'double' is declared as '<java/lang/Double.<init>>'

Conjecture: The evidence that the function 'FUNCCompare()' make method call to 'string.equal()', implies that this function might be comparing the two string passed as a parameters. The instance initializations of local variable of type double does not provide any logical support to this assumption.

Final Conclusion: One can conclude that this class must be providing the comparison functionality.

Confidence Rank: Medium.
Value Ranking: Low

d) **Class Name:** **procentOf**
Assumption: The word 'procentOf' seems like 'percent of', but still we don't have enough evidence to make any assumptions about this class's functionality.

Total Fields: NONE

Total Methods: One

1. public static
   FUNCprocentOf(java.lang.Double,java.lang.Double,java.lang.Object[])
   returns: java.lang.Object

Assumption: Two of the parameters passed to this function are of type double. One can make weak assumption that this function might be calculating percentage of these parameters.

Method Bytecode:
- Function 'FUNCprocentOf()' is calling:
  - java.lang.Double.doubleValue() returns: double
- Other Information:
  - Two local variables of type double are loaded on the stack.
  - A constant of type double is found to have assigned value of 100.0
  - Two arithmetical operations, division and multiplication are found to be used after the above constant declaration.
  - One instance of java.lang.Double is and loaded. Method returns reference to it.

Conjecture: The presence of division and multiplication operations, and constant declaration of value 100.0 provides enough evidence to make conjuncture about this method's functionality, that it should be calculating percentage value for the two parameters of type double.

Final Conclusion: It seems that the class 'procentOf' is providing functionality of calculating percentage for this Open Calculator application.

Confidence Rank: High
Value Ranking: Low

e) **Class Name**: **rand**
Assumption: The name of this class does not provide evidence to make strong assumptions about its purpose and function.
Total Fields: NONE
Total Methods: One

1. public static FUNCran**d**(java.lang.Object[]) returns: java.lang.Object
Assumption: The name and signature of this method does not suggest anything about its functionality.

Method Bytecode:
- Function 'FUNCrand()' calls methods listed below:
  - java.lang.Math.sin(doble) returns: double

- **Other Information**:
  - Code length of this method is 14.
  - A local constant is declared and assigned value of 12.0
  - This method performs a trigonometry function sine on a double variable.

Conjecture: We can make a conjecture that this function seems to perform a trigonometry sine function. Code length of this method is found to be 14 which very less and there are no other operations except the sine function are found in this method.

Final Conclusion: The above information, which is found by exploiting its bytecode, suggests that this class might be providing trigonometric sine functionality for the open calculator. We can not confirm about the overall purpose of this class, since we couldn't found enough reasons for the declared constant of value 12.0.

Confidence Rank:
Value Ranking: Low

f) **Class Name**: **random**
Assumption: The name of the class does not provide enough information to assume about its purpose and functionality.

Total Fields: NONE
Total Methods: One

1. public static FUNCrandom(java.lang.Object[])
returns: java.lang.Object

Assumption: The name of the function does not tell us enough to assume its functionality.

Method Bytecode:

- The function 'FUNCrand()' makes other method calls as follow:
  - java.lang.Math.randon() returns: double
- **Other Information**:
  - The code length of this method is 11.
  - The function returns a reference to a data type double.

Conjecture: The total code length of this method is very less, and it is making a method call to 'java.lang.Math.rand()' function. This provides us enough information to conclude that this method generates a random number.

Final Conclusion: One can conclude confidently about the purpose of this class. This class provides a functionality of random number generation.

Confidence Rank: High
Value Ranking: Low

g) **Class Name**: **sumIntegral**
Assumption: The name of the class does not provide enough information to make strong assumptions about its purpose and functionality.

Total Fields: NONE
Total Methods: One

1. public static FUNCsumIntegral(string,string,java.lang.Double,java.lang.Double,java.lang.Double[])
returns: java.lang.Object

Assumption: One can make an assumption that this function might be performing integral function. But, Its name and the signature does not provide enough evidence to support this assumption.

Method Bytecode:

- The function ‘FUNCsumIntegral() make following method calls:
  - java.lang.String.charAt (int) returns: char
  - opencalculator.api.OpenCalculate.getVariable(int) returns: double
  - opencalculator.api.OpenCalculate.count(string) returns: double
  - opencalculator.api.OpenCalculate.setVariable(java.lang.Double) returns: void

- **Other Information**:
  - Constant declaration of the value “1000000.0” This constant is declared between the two arithmetic operations ‘subtraction’ and ‘multiplication’.
  - Another constant of the value “5.0E-7” is loaded on the stack before the function ‘addition’.
  - A constant of the value “1.0E-6” is loaded on the stack inside a ‘if greater then or equal to’ condition, followed by another ‘multiplication’ and ‘addition’ functions.
  - A constant declaration of the value “1.0E-6” and a ‘addition’ is found between the invocation of ‘openCalculate.getVariable()’ and ‘openCalculate.setVariable()’
  - A string “fel i funktioen som foljer med sumIntegral” is found before the final return statement. The word-to-word English translation of this string is “error at/for functions which/who/like UNKNOWN with/by sumIntegral”.

Discussion and Conjecture: The chartAt() method returns the character at the specified index. After this statement an instance of ‘api.OpenCalculate’ is created, followed by some constant declarations of type double and mathematical operations. The ‘OpenCalculate.getVariable()’ and ‘OpenCalculate.setVairable()’ are the getter and setter methods for the field ‘Variables’ of type ‘java.lang.Object[]’ and declared in the class ‘api.OpenCalculate’. All this information does not provide the exact picture of this function. It seems that this function is performing some mathematical operations and calling methods from the class ‘api.OpenCalculate’. The string found before return statements seems to be an error message display for the function ‘sumIntegral’. All these information is not sufficient to make conjuncture about the functionality of ‘FUNCsumIntegral()’.

Final Conclusion: This class may be performing some arithmetical functions for the Open Calculator application. Since we could not locate any ‘java.lang.Math’ library functions that perform integration, we can not conclude that this class provided integration functionality.

Confidence Rank: Low
Value Ranking: High

h) **Class Name**: **timeMs**
Assumption: One can make an assumption that this class may be providing functions to calculate time intervals in milliseconds
Total Fields: NONE
Total Methods: One

1. public static FUNCtimeMs (java.lang.Object[]) returns: java.lang.Object
Assumption: The function's name suggests that it might be calculating the time / time interval in milliseconds. But its signature does not provide any extra information to support this assumption.
Method Bytecode:

- The 'FUNCtimeMs()' is making following method calls:
  - java.lang.System.currentTimeMillis() returns: long

- **Other Information**:
  - Total code length of this method is 12.

Discussion and Conjecture: The code length of this method is significantly less, which makes it easy to assess it. The only method call found is 'System.currentTimeMillis()', which returns the current time in milliseconds. This is the sufficient evidence to conclude that the function 'FUNCtimeMs()' returns current time in milliseconds.

Final Conclusion: The purpose of this class is to provide the current time in milliseconds.

Confidence Rank: High

Value Ranking: Low

Level 0 Assessment based on the hierarchies and relationship amongst the classes found under the 'API.func'package:

Hierarchies: No hierarchies are found in this package.

Dependencies and Relationships: There are no relationships amongst all the eight classes. The only class 'sumIntegral' has dependency relationship with an external class 'opencalcualtor.api.OpenCalculate'.

Discussion and Conjecture: Since all the classes except the 'sumIntegral' classes are not dependent or related to any other class of this application, one can be sure that this class must be having independent or stand-alone functionality. The class 'sumIntegral' is calling three methods of the 'opencalculator.api.OpenCalculate' class to accomplish its purpose.

Level 1 Assessment for the API.func package.

Package Name: API.func
Assumption: The package 'func' is found under the 'API' package. API is a collection of programmatic elements (set of routines, protocols, tools, etc.) that is called by other piece of software. It is technique of abstraction. Since this package is found under the API package, one can assume that it must be providing a set of functions that are used by other packages of the Open Calculator application.

Final Conclusion: The 'Level 0' assessment of the classes and their dependencies has provided some significant evidence so that one can conclude that the 'API.func' package provides a set of functions for the Open Calculator application. The partial list of functions is return current time in milliseconds, calculate percentage, perform comparison, random number generation, etc. But this does not look like a complete set of arithmetical or other functions that any typical calculator application provides. A calculator should have a rich of functions.
One can make following assumptions and conjecture based on this assessment:

- The current version of the Open Calculator might be still under construction and the developers are still adding other functionalities.

Confidence Rank: High

Value Ranking: Medium

II. Level 0 Assessment Results for API package:

a) **Class Name**: **funcRunner**
Assumption: The name of the class 'funcRunner' suggests that it might be providing functionalities necessary to run the function defined in the 'API.func' class or any other functions.

Total Fields: One

1. Variables java.lang.Objects[]
Assumption: The name and the data type of this field do not imply anything about its purpose.

Total Methods: Four

1. protected findClass (string) returns: java.lang.Class
Assumption: The name of this function implies that it might be finding the class or its path. It returns java.lang.Class type, which may represent array or any primitive Java types (boolean, byte, short, int, etc.). The string parameter passed to it may be t he name of the class.

Method Bytecode:

- The method ‘findClass’ make following method class:
  - java.lang.String.replace (char, char) returns: string
  - java.lang.StringBuilder.append (string) returns: java.lang.StringBuilder
  - java.io.FileInputStream.available () returns: int
  - java.io.FileInputStream.read (byte []) returns: int
  - opencalculator.api.funcRunner.defineClass (string,byte [],int,int) returns: java.lang.Class
  - java.io.FileInputStream.close () returns: close

- **Other Information**:
  - A string constant is declared and it is assigned a value “.class”. The ‘StringBuilder.append()’ function is called before and after this string declaration.
  - A ‘arraylength’ instruction is found before the function ‘opencalculator.api.funcRunner.defineClass()’ is called.
  - Two ‘if equals’ conditions are found for the string constants which are loaded on the stack, and there are two ‘FileInputStream.close()’ function calls are found for each ‘if equals’ statements.
  - A ‘java.lang.ClassNotFound’ exception is thrown.

Discussion:

- The ‘String.replace()’ returns new string resulting from replacing all the occurrences of an old character with the new character. The ‘StringBuilder.append()’ function is called after the declaration of a string with the value ‘.class’, which implies that “.class” is appended after a string which can be the name of a class. But it is difficult to guess why the method ‘String.replace()’ is called before appending “.class” to a string.
- The method ‘FileInputStream.available()’ returns the number of bytes that can be read from this file stream. The ‘anewarray’ instruction is found after this method call and this instruction is used to create an array of numeric type. It seems that an array is been created of the size returned by ‘FileInputStream.available()’ method.
- The method call ‘FileInputStream.read (byte [] b)’ reads up to b.length byte of data from this input stream into the array which has been created before.
- There is no method with the name ‘defineClass()’ in the ‘funcRunner’ class, so it is difficult to assume what it is used for.
- The ‘if equals’ conditions are comparing two strings, and a reference to the class is returned for a successful comparison. It seems that if the class is successfully found then its contents are retuned as a reference to it. It is difficult to judge that why there are two declarations of the ‘if equals’ conditions.
- The ‘java.lang.ClassNotFound’ exception might be thrown when the class is not found in the specified class path.

Conjecture: Once can make a conjecture about this method's overall functionality that it accepts the name of the class as a string parameter, finds it, and reads it byte by byte. But, it is difficult to judge the purpose of method calls 'String.replace()', 'opencalculator.api.funcRunner.defineClass()', and two 'if equals' conditions.

Confidence Rank: Medium
Value Ranking: Low

2. private getParameters (string) returns: java.lang.Object[]
Assumption: The name suggests that these methods return some type of parameters, but it is difficult to judge what parameters are these.

Method Bytecode:

- The method 'getParameters()' make following method calls:
  - java.lang.String.length() returns: int
  - java.lang.String.indexOf (int,int) returns: int
  - java.lang.String.substring (int,int) returns: string
  - java.util.Vector.add (java.lang.Object) returns: boolean
  - java.util.Vector.size() returns: int
  - java.util.Vector.elementAt (int) returns: Object
  - opencalculator.api.OpenCalculateKomando.run(string) returns: Object

- **Other Information**:
  - A 'addition' operation on integer is called before the 'String.indexOf()' is invoked.
  - The 'String.indexOf()', 'string.substring()', and 'Vector.add()' methods calls are found inside a loop.
  - A 'anewarray' statement called after the method call 'Vector.size()'.
  - The 'Variables' field is accessed when the 'opencalculator.api.OpenCalculateKomando' class is initiated.
  - Method 'Vector.size()' is called followed by a for loop with an increment counter. The 'Vector.elementAt()', 'OpenCalculateKomando.run()' method calls are found in this loop.
  - An array of 'java.lang.Object' is created after this 'for' loop.
  - Another 'for loop is found in which the 'funcRunner.variables'[sic] field is accessed.

Discussion: One can make the following conjecture based on the information gathered from the bytecode.

- The length of the string parameter passed to this method is determined. The next 'if' condition checks for invalid value of this parameter.
- It seems that the 'indexOf()' function returns the index of the specified character found in the string parameters passed to this method.
- The next 'for' loop has some string manipulation functions and the

'Vector.add()' method call. The 'java.util.Vector' implements a growable array of objects. One can assume that this 'for' loop extracts some sub-strings and appends them to the end of the current instance of the object by calling .Vector.add()' method.

- It looks like that the another loop is existed inside the first loop and it has another string manipulation functions like 'String.length()', 'string.substring()' and the 'Vector.add()' function
- The method call 'Vector.size()' returns the number of components in the instance of the current Vector. The 'anewarray' instruction is found after the 'vector.size()' method call. Because of this bytecode sequence, one can assume that the new array is created of the size returned by the 'Vector.size()' method. The class member field 'Variables' is accessed after the 'anewarray' statement. It can be inferred that the field 'Variables' is assigned values of the array which is created from the Vector components.
- The next loop found to have method calls to 'Vector.elementAt()', 'OpenCalcualteKomando.run()'. The 'OpenCalcualteKomando.run()' takes one parameter of type string and the 'Vector.elementAt()' returns the component at the specified index. Based on these facts, one can assume that the 'OpenCalculateKomando.run()' method is called for each component returned by the 'Vector.elementAt()' method.
- The method returns a variable of the type 'java.lang.Object[]'. This method's bytecode and its return type imply that this method might be returning the array of the strings collected iteratively by the string manipulation functions.

Conjecture: After the complete assessment of this method's bytecode, one can make a conjecture that this method is parsing the string, storing each sub-strings in the 'Variables' field, and is calling the 'OpenCalculateKomando.run()' method by passing each string as a parameter.

Confidence Rank: Medium
Value Ranking: Low

3. private getTypes (java.lang.Object[]) returns: java.lang.Class[]
Assumption: One can assume that this function might be returning some type of parameters.

Method Bytecode:

- The method 'getTypes()' makes following method calls:
  - java.lang.Object.getClass() returns: java.lang.Class

- **Other Information**:
  - The code length of this method is 31.
  - The 'arraylength' instruction is found before the 'anewarray' instruction.

- Another ‘arraylength’ instruction is found followed by a loop. The method call ‘Object.getClass()’ is found within the loop code.

Discussion and Conjecture: One can assume that the ‘arraylength’ should be finding the array length of the parameter passed to this method. The ‘Object.getClass()’ method returns the runtime class of an Object. In this case, it must be returning the Class for each object passed as a parameter. This implies that the method ‘getTypes()’ must be returning the ‘java.lang.Class []’ for the ‘java.lang.Object []’ parameter.

Confidence Rank: High
Value Ranking: Low

4. public runFunc(string) returns: java.lang.Object
Assumption: The name of this method suggests that it may be running the function of the name passed to it as a parameter.

Method Bytecode:

- The function ‘runFunc()’ makes the following method calls:
  - java.lang.String.indexOf (int) returns: int
  - java.lang.String.substring (int,int) returns: string
  - java.lang.StringBuilder.append (string) returns: java.lang.StringBuilder
  - opencalculator.api.funcRunner.findClass (string) returns: java.lang.Class
  - java.lang.String.substring (int,int) returns: string
  - opencalculator.api.funcRunner.getParameters (string) returns: Objects []
  - opencalculator.api.funcRunner.getTypes (Objects[]) returns: Class []
  - java.lang.Class.getMethod (string,java.lang.Class[]) returns: java.lang.Method
  - java.lang.reflect.Method.invoke (Object, Object[]) returns: Object
  - java.lang.Exception.getMessage () returns: string

- **Other Information**:
  - Stack loads a string constant of value “opencalculator.api.func.” followed by a ‘StringBuilder.append()’ function.
  - Another string constant if found of the value “FUNC”.
  - A string declaration is found of the value “Funktionen hittades ej”. It English translation is “Function UNKNOWN not”.
  - It is followed by ‘java.lang.Exception.<init>>’ and a ‘athrow’ instruction.

Discussion and Conjecture:

- The method ‘runFunc()’ performs some string operations on the parameters passed to it and then a string “opencalculator.api.func.” is appended to it and tries to build a string like “opencalculator.api.func.xxxxxx”. This implies that the method is

accessing classes from the “opencalcualtor.api.func” package.

- The string constant “FUNC” is appended to another string. Interestingly the word “FUNC” is found in all the methods declared in the ‘api.func’ package.
- Then the method call ‘funcRunner.findClass()’ is searching for that particular class and reads its contents byte by byte.
- The method calls ‘funcRunner.getParameters()’ and ‘funcRunner.getTypes()’ suggests that the parameters and their types are being accessed from that class or the class file in the ‘opencalculator.api.func’ directory.
- The function ‘java.lang.reflect.Method.invoke()’ is called after the ‘java.lang.Class.getMethod()’ function is invoked. It is evident that the method with a string “FUNC” in its name and which is declared in “opencalculator.api.func.xxxxxxx’ class is retrieved and invoked using the ‘java.lang.reflect.Method.invoke()’ function.
- The exception message is displayed with the string “Function not UNKNOWN”. The word UNKNOWN is used since the English translation of the word ‘hittades is not found. This is a non-English string found in the bytecode along with the code dealing with the ‘java.lang.Exception’ class.

Final Conclusion for the class ‘funcRunner’: The above assessment and discussion suggests that this class has functions necessary for finding and reading a class file, retrieving its parameters and types, and running the function declared in that class file. The ultimate purpose of this class seems to run the function definitions declared in the ‘opencalculator.api.func’ package.

Confidence Rank: Medium
Value Ranking: High

b) **Interface Name**: **ioAble**
Assumption: The name of this interface suggests that its purpose is to enable the input-output functions for this application.

Total Fields: NONE
Total Methods: Three

1. *public abstract get() returns: string*
Assumption: Its name and signature suggests that this might be a function definition for retrieving some string.

2. *public abstract print(java.lang.Object) returns: void*
Assumption: It implies that this method definition is to enable print functionality.

3. *public abstract println(java.lang.Object) returns: void*
Assumption: It might be defining functionality to print and then terminate the line.

Conjecture: It is difficult to make a solid conjecture about the purpose of this interface unless one can get more information about the classes that are extending it. Although, one can assume that it is providing function definition for printing operations.

Confidence Rank: High
Value Ranking: Low

c) ***Class Name:*** ***OCError***
Assumption: The fact that this is an abstract class and its name suggests that it might be defining functions to perform 'Open Calculator Error" operations.

Total Fields: One

1. *typ*[sic] *int*
Assumption: The name and data type of this field implies that it might be defining some kind of number (or type) assignments for each errors

Total Methods: One

1. *public getTypeOfError () returns: int*
Assumption: The method name and its signature implies that it must be retrieving the 'typ'[sic] variable.

Method Bytecode:
- No external method calls.

- **Other Information**:
  - a 'getfield' instruction is found with the 'opencaclulator.api.OCError.typ[sic]' string.

Discussion and Conjecture:
- One can easily conclude that it is a getter method of this class and it is retrieving the value of the 'typ' variable.

Final Conclusion: The only thing one can tell about this abstract class is that it declares a variable for the error types. Further information gathered from the bytecode assessment of the classes that inherits from it can provide more idea.

Confidence Rank: Medium
Value Ranking: Low

d) **Class Name**: **OCPError**
Assumption: One can assume that this class must be having operations to define a type of OCError.

Total Fields: Two

1. Line int
Assumption: This seems to be a line number.

2. typOfProgramError[sic] int
Assumption: This variable might be assigning type (or numbers) to the program errors.

Total Methods: Two

1. public getType() returns: int
Assumption: This method seems like the getter method that retrieves value of the 'typOfProgramError'[sic] field.

Method Bytecode:
- No external method calls.

- **Other Information**:
  - A 'getfield' instruction is found with 'opencalculator.api.OCPError.typOfProgramError'.

Discussion and Conjecture: This method retrieves the value of 'typOfProgramError' field.

1. public getLine() returns: int
Assumption: This method seems like the getter method that retrieves value of the 'Line' field.

Method Bytecode:
- No external method calls.

- **Other Information**:
  - A 'getfield' instruction is found with 'opencalculator.api.OCPError.Line'.

Discussion and Conjecture: This method retrieves the value of 'Line' field.

Final Conclusion: The class purpose of the OCPError' class is to define operations for the program errors that can occur while the application is running and to return the number where the error occurred.

Confidence Rank: High
Value Ranking: Low

d) **Class Name**: **OCprogramError**
Assumption: This purpose of this class might be to provide functions that defines program errors for the Open Calculator application.

Total Fields: Two
1. Line int
Assumption: Line number where the error has occurred.

2. error opencalculator.api.OCError
Assumption: A instance of the class 'opencalculator.api.OCError' class.

Total Methods: Two

1. public getError() opencalculator.api.OCError
Assumption: Retrieves value of the 'error' field.

Method Bytecode:
- No external method calls.

- **Other Information**:
- A 'getfield' instruction is found with 'opencalculator.api.OCPError.error'.

Discussion and Conjecture: This method retrieves the value of the 'error' field which is an instance of the 'OCError' class.

Confidence Rank: High
Value Ranking: Low

2. public getLine() int
Assumption: Retrieves value of the ''Line' field.

Method Bytecode:
- No external method calls.

- **Other Information**:
- A 'getfield' instruction is found with 'opencalculator.api.OCPError.Line'.

Discussion and Conjecture: This method retrieves the value of the 'line' field which seems to be the line number where the error might have occurred. This may be the line number of some file, or some program. It is difficult to conclude whose line number this would be.

Final Conclusion: The class 'OCprogramError' is a sub-class of 'OCError'. This fact and the above assessment implies that its purpose is to provide the functions to define program error that can occur during the Open Calculator application operations.

Confidence Rank: High
Value Ranking: Low

e) Class Name: OCSyntaxError

Assumption: This class might be providing functionalities for syntax errors related to this application.

Total Fields: One

1. typOfSyntaxError[sic] int

Assumption: This variable might be assigning numbers to various types of syntax errors

Total Methods: One

1. public getType() returns: int

Assumption: This is the getter method that retrieves value of the 'typOfSyntaxError'[sic] field.

Method Bytecode:

- No external methods are called.
-

- **Other information:**
  - Code length is 5
  - A 'getfield' instruction is found with 'opencalculator.api.OCSyntaxError.typOfSyntaxError'

Conjecture: This method returns the value of the field 'typOfSyntaxError'. In other words, one can assume that this method must be returning the type of the syntax error that has occurred.

Final Conclusion: The class 'OCSyntaxError' represents a type of errors that can occur in the Open Calculator application. Its purpose is to encapsulate functionalities related to the syntax error. The name of this class implies that the Open Calculator might be allowing the user to create programs, since a syntax error can only occur while the application is trying to compile and run the code written by the user.

Confidence Rank: Medium
Value Ranking: Low

f) Class Name: **OCVariableError**

Assumption: This class might be encapsulating functionalities for some type of variable errors.

Total Fields: One

1. variabel[sic] int

Assumption: This field might be assigning numbers to all the variables or to all the types of possible variable errors. The incorrect spelling of this field is may be because the developers of this application seem to be of non-English origin, or it may be a mere spelling mistake made by the developers.

Total Methods: One

1. public getVariable() returns: int

Assumption: This method returns the value of the 'variabel'[sic] field.

Method Bytecode: No method calls.

- **Other information:**
  - The code length of this method is 5 which is significantly less and makes the assessment easy.

Conjecture and Discussion: This method merely returns the value of the number that might be assigned to the variables or to the variable errors.

Final Conclusion: This class encapsulates functionalities to represents the variable errors which are a type of Open Calculator errors. But, at this stage it is difficult to judge that what types of variable error may occur in this application.

Confidence Rank: Medium
Value Ranking: Low

g) Class Name: **OpenCalculate**

Assumption: The name of the class represents the name of the application which is the Open Calculator. One can assume that this class might be entry point when the application is started.

Total Fields: Two

1. Variables java.lang.Objects []

Assumption: This field might be representing the array of some type of variables.

2. angle int

Assumption: This field might be representing the value of an angle.

Total Methods: Ten

1. private HittaMinustal (opencalculate.api.OpenCalculateSatts)
   returns: opencalculate.api.OpenCalculateSatts

Assumption: The English translation of the word 'HittaMinustal" is "find minus number. So, this method must be finding some 'minus number', but it is difficult to judge about its actual functionality.

Method Bytecode:

- The method 'HittaMinustal' makes following external method calls.
  - opencalculator.api.OpenCalculateSatts.getIdentifier(int) returns: int
  - opencalculator.api.OpenCalculateSatts.getValue(int) returns: double
  - opencalculator.api.OpenCalculateSatts.setNumber(int,double) returns: void
  - opencalculator.api.OpenCalculateSatts.remove(int) returns: void

- **Other information:**
  - Two condition branches – 'if not equals' and 'if equals' are found.
  - A string '0giltif minusoperator' is found. Its word-by-word English translation is "0 valid minus operator".
  - The above string and a 'java.lang.Exception' declaration is found in the second conditional branch 'if equals', which is comparing the integer values.
  - A 'subtract' operation is found between the method calls 'opencalculator.api.OpenCalculateSatts.getValue()' and the 'opencalculator.api.OpenCalculateSatts.setNumber()'.

Conjecture and Discussion:

- This method accepts a parameter of type 'opencalculator.api.OpenCalculateSatts' and returns of the same type. It implies that this method must be performing some operations on the object of the 'opencalculator.api.OpenCalculateSatts' class.
- In brief, all the method calls from the class 'OpenCalculateSatts' does not tell a lot about what it is doing, except that they are accessing and performing some mathematical operations on the field 'OpenCalculateSatts.Elementen'.
- Based on these class methods, one can assume that this function is finding some identifier and the value, setting some number, and then removing it. All these operations are being performed on a object of 'java.util.Vector' class and the 'Elementen' field of the 'OpenCalculateSatts' class.
- The 'opencalculator.api.OpenCalculateSatts.getIdentifier()' is called in the first 'if not equals' condition branch. The 'java.lang.Exception', and the string '0giltif minusoperator' (English translation: '0 valid minus operator') are found in the second condition 'if equals'. Based on this, one can assume that if the integer values does not matches, then the 'opencalculator.api.OpenCalculateSatts.getIdentifier()' is called and else the exception is thrown with a message '0 valid minus operator'.

- It is very difficult to conclude about what this method is doing, except one can make the above assumptions about what might be happening inside the method.

Confidence Rank: Low
Value Ranking: Low

2. private UtforFunktioner (opencalculator.api.OpenCalculateSatts)
returns: opencalculator.api.OpenCalculateSatts

Assumption: The English translation of the word 'UtforFunktioner' is 'out for functions'. But, this does not provide enough information about its functionality.

Method Bytecode:

- The following method calls are found:
  - opencalculator.api.OpenCalculateSatts.size() returns: int
  - opencalculator.api.OpenCalculateSatts.getIdentifier(int) returns: int
  - opencalculator.api.OpenCalculateSatts.getValue(int) returns: double
  - opencalculator.api.OpenCalculate.getFunctionValue (int,double) returns: double
  - opencalculator.api.OpenCalculateSatts.setNumber(int,double) returns: void
  - opencalculator.api.OpenCalculateSatts.remove(int) returns: void

- **Other information:**
  - A conditional branch 'if greater then', and two 'loops are found since this conditional branch.
  - The 'opencalculator.api.OpenCalculateSatts.getIdentifier()' is found being called at regular intervals.
  - A string 'Inget tal angivet till funktion' is found. Its English translation is 'not/no number declared for function'.
  - An instance of the class 'OCSyntaxError' is created where the above string declaration is found.

Conjecture and Discussion:

- The method 'opencalculator.api.OpenCalculateSatts.size()' returns the size of the field 'Elementen' of the class 'OpenCalculateSatts' in terms of 'java.util.Vector.size()'. The method 'opencalculator.api.OpenCalculateSatts.getIdentifier()' performs some operations on the field 'Elementen' and returns an integer value, which might be some type if identifier value. The method 'opencalculator.api.OpenCalculateSatts.getValue()' seems to return the value at the specified index from the field 'Element' which is of the type 'java.util.Vector'. The method 'opencalculator.api.OpenCalculate.getFunctionValue()' is performing some trigonometric operations on the field 'opencalculator.api.OpenCalculate.angle' inside a switch loop.

- The error message ‘not/no number declared for function’ and the object instantiation of the class ‘OCSyntaxError’ implies that this method seems to be performing some operations on some functions. Even with all these information, it is difficult to conclude about the overall functionality of this method.

Confidence Rank: Low
Value Ranking: Low

3. private UtforOperator (opencalculator.api.OpenCalculateSatts,int,int)
returns: opencalculator.api.OpenCalculateSatts

Assumption: The English translation of the word ‘UtforOperator’ is ‘out for operators’. One can assume that this method might be searching for some operators.

Method Bytecode:

- It is making following method calls:
  - opencalculator.api.OpenCalculateSatts.getValue(int)
    returns: double
  - opencalculator.api.OpenCalculate.getOperationValue(int,double,double)
    returns: double
  - opencalculator.api.OpenCalculateSatts.setNumber(int,double)
    returns: void
  - opencalculator.api.OpenCalculateSatts.remove(int)
    returns: void

- **Other information:**
  - A conditional branch ‘if greater then’, and two ‘loops are found since this conditional branch.
  - The ‘opencalculator.api.OpenCalculateSatts.getIdentifier()’ is found being called at regular intervals.
  - A string ‘Ett eller flera tal till operator saknas’ is found. Its English translation is ‘one or/nor many numbers for operators lack/wants’.
  - An instance of the class ‘OCSyntaxError’ is created where the above string declaration is found.

Conjecture and Discussion:

- The bytecode structure of this method is similar to the method ‘UtforFunktioner()’, except this method is calling ‘opencalculator.api.OpenCalculate.getOperationValue()’ where is the ‘UtforFunktioner()’ is calling ‘opencalculator.api.OpenCalculate.getFunctionValue()’ function.
- The method ‘opencalculator.api.OpenCalculate. getOperationValue ()’ has a switch statement with some ‘if..else’ conditional loops.

The‘java.lang.Math.pow() is the only method call found.
- The string that may be displayed as a message is ‘one or/nor many numbers for operators lack/wants’, and this does not imply anything significant about this method.
- From this method’s bytecode and the assessment of the ‘UtforFunktioner()’, one can make a conjecture that this method this method performs some function on the operators. It is difficult to judge what these operators are.

Confidence Rank: Low
Value Ranking: Low

4. public count (string) returns: double

Assumption: The name of this method suggests that it must be returning some type of count, but its signature does not provide any useful information to support this assumption.

Method Bytecode:

- It make the following method calls:
  - opencalculate.api.OpenCalculate.getValue (opencalculator.api.OpenCalculateSatts) returns: double

- **Other information:**
  - The code length of this method is 19.
  - The value of the field ‘opencalcualtor.api.OpenCalculateSatts.Variables’[sic] is accessed.

Conjecture and Discussion:

- First an object of the class ‘OpenCalculateSatts’ is instantiated and then the value of the field ‘Variables’ is accessed which returns an array of type ‘java.lang.Object[]’.
- Then the method ‘opencalculator.api.OpenCalculate.getValue()’ is invoked and the instance of the class ‘OpenCalculateSatts’ is passed as an argument.
- The method ‘opencalculator.api.OpenCalculate.getValue()’ returns a value of type double and this is what retuned by the method ‘count()’.
- So, the ‘count()’ method does nothing but calls the ‘opencalculator.api.OpenCalculate.getValue()’ method to accomplish its purpose. But, it is difficult to conclude the exact functionality of ‘count ()’ method before the complete assessment of ‘opencalculator.api.OpenCalculate.getValue()’ is done.

Confidence Rank: Medium
Value Ranking: Low

5. private countUtanParantes (opencalculator.api.OpenCalculateSatts) returns: java.lang.Double

Assumption: The English translation of the word 'countUtanParantes' is 'count without/but parenthesis'. Its name implies that this method might be returning some value related to the parenthesis count.

Method Bytecode:

- It makes following method calls.
- opencalculator.api.OpenCalculate.UtforFunktioner(opencalculator.api.OpenCalculateSatts) returns: opencalculator.api.OpenCalculateSatts
- opencalculator.api.OpenCalculate.HittaMinustal(opencalculator.api.OpenCalculateSatts) returns: opencalculator.api.OpenCalculateSatts
- opencalculator.api.OpenCalculate.UtforOperator(opencalculator.api.OpenCalculateSatts,int,int) returns: opencalculator.api.OpenCalculateSatts
- opencalculator.api.OpenCalculateSatts.size() returns: int
- opencalculator.api.OpenCalculateSatts.getValue (int) returns: double

- **Other information:**
  - One conditional branch of 'if less the equals' is found with the object of the 'OCSyntaxError' class and a message string 'Opration saknas' is found. The English translation of this message is 'Operation missing'
  - This method returns a double value which is calculated by the 'opencalculator.api.OpenCalculate.getValue()' method.

Conjecture and Discussion:

- The long length of this method's bytecode makes it difficult to assess its internal operations.
- The method calls made by this method implies that this method is operating on results returned by the 'opencalculator.api.OpenCalculate.HittaMinustal()' and 'opencalculator.api.OpenCalculate()' methods. It is interesting to note that the method 'opencalculator.api.OpenCalculate.UtforOperator()' is invoked seven times whereas the other two methods are called only once.
- The purpose of method calls to 'opencalculator.api.OpenCalculateSatts.size()' and 'opencalculator.api.OpenCalculateSatts.getValue()' is difficult to judge.
- The error message 'operation missing' refers to the failure condition of this method and it implies that this method might be searching for some operators and in case they are not found then this message is displayed along with the object instantiation of the 'OCSyntaxError' class.

Confidence Rank: Low
Value Ranking: Low

6. private getFunctionValue (int,double) returns: double

Assumption: The name and the signature of this method implies that it might execute some function and returning its results.

Method Bytecode:

- Following methods calls are found in the bytecode:
  - java.lang.Math.sin (double) returns: double
  - java.lang.Math.toRadians (double) returns: double
  - java.lang.Math.cos (double) returns: double
  - java.lang.Math.tan (double) returns: double
  - java.lang.Math.asin (double) returns: double
  - java.lang.Math.atan (double) returns: double
  - java.lang.Math.acos (double) returns: double
  - java.lang.Math.log (double) returns: double

- **Other information:**
  - A switch conditional branch is found.
  - The field 'opencalculator.api.OpenCalculate.angle' is accessed in each condition of the switch loop.
  - Each condition of the switch loop has one 'if not equals' conditional branch and the trigonometric functions are called in this 'if' condition.
  - A constant with the value "10.0" is found when the function 'java.lang.Math.log()' is called.
  - A string 'Boolean operator utan boolean varden' is found whose English translation is 'Boolean operator in/at/for/on boolean UNKNOWN'.
  - An object of the class 'OCSyntaxError' is instantiated.

Conjecture and Discussion:

- The trigonometric functions are calculating sine, cosine, tan, arccosine, arctangent, and arcsine of the 'opencalculator.api.OpenCalculate.angle' field. These trigonometric values are calculated inside each switch condition and the value of 'opencalculator.api.OpenCalculate.angle' is converted into radians before its trig values are calculated.
- The constant of the value <10.0> is found along with the method call 'java.lang.Math.log', which implies that, the logarithm to the base ten is being calculated.
- The error message 'Boolean operator in/at/for/on boolean UNKNOWN' found with the object instantiation of the class 'OCSyntaxError' does provide any clue about the erroneous condition of this method.
- From the above discussion and bytecode assessment, one can conclude that this method provides trigonometric operations for the Open Calculator application. But it is difficult to judge the exact purpose of 'getFunctionValue()' method.

Confidence Rank: Medium
Value Ranking: High

7. private getOperationValue (int,double,double) returns: double
Assumption: This method might be calculating the value of some operations.

Method Bytecode:

- Following method calls are made:
  - java.lang.Math.pow(double,double) returns: double

- **Other information:**
  - A switch conditional branch is found.
  - A string message is found 'Division med 0' whose English translation is 'Division by 0'.
  - Mathematical operations like, subtraction, addition, multiplication, division, 'java.lang.Math.pow()' are found in each switch condition.
  - Some 'if' conditional branches found in the next set of switch conditions.
  - A string 'Boolean operator utan boolean varden' is found declared two times, whose English translation is 'Boolean operator in/at/for/on boolean UNKNOWN'.
  - An object of the class 'OCSyntaxError' is instantiated twice along with the above text message.

Conjecture and Discussion:

- Each condition of th**e** switch branch performs basic arithmetic operations. The next switch conditions has 'if greater then equals', 'if less then equals', 'if less then', 'if greater then', 'if not equals', 'if equals'. It implies that this method performs the basic arithmetical and logical operations for the Open Calculator application.
- The exact purpose of the error message is difficult to judge.

Confidence Rank: Medium
Value Ranking: High

8. private getValue (opencalculator.api.OpenCalculateSatts)
returns: double

Assumption: This method might be returning some value of an object of 'opencalculator.api.OpenCalculateSatts' class.

Method Bytecode:

- 'getValue()' is calling:
  - opencalculator.api.OpenCalculateSatts.size() returns: size
  - opencalculator.api.OpenCalculateSatts.getInersteParentes()
    returns: opencalculator.api.OpenCalculateSatts
  - opencalculator.api.OpenCalculate.countUtanParentes (opencalculator.api.OpenCalculateSatts) returns: java.lang.Double
  - opencalculator.api.OpenCalculateSatts.delInersteParentes (java.lang.Double) returns: void

- java.lang.Exception() returns: string
- java.lang.String,equalsIgnoreCase (string) returns: boolean
- opencalculator.api.OpenCalculateSatts.getIdentifier(int) returns: int
- opencalculator.api.OpenCalculateSatts.getValue(int) returns: double

- **Other information:**
  - The first loop contains method calls 'opencalculator.api.OpenCalculateSatts.getInersteParentes()', 'opencalculator.api.OpenCalculate.countUtanParentes()', and 'opencalculator.api.OpenCalculateSatts.delInersteParentes()' is sequence.
  - A error message 'Ingen Parantes' is found with 'java.lang.Exception.getMessage()'. Its English translation is 'No parenthesis'.
  - The next 'if not equals' condition has an object instantiation of the class 'OCSyntaxError' and a text message 'Parantes ERROR', whose English translation is 'parenthesis ERROR'.
  - The next 'if compare equals' condition has an object instantiation of the class 'OCSyntaxError' and a text message 'Inget gilltigt utryck', whose English translation is 'None UNKNOWN UNKNOWN'.
  - The method 'opencalculator.api.OpenCalculateSatts.getValue()' is being called before the final return statement.

Conjecture and Discussion:
- The method 'getValue()' actually seems to calling 'opencalculator.api.OpenCalculateSatts.getValue()' for the parameters of the type 'opencalculator.api.OpenCalculateSatts'.
- The size of the field 'Elementen' of the object of 'OpenCalculateSatts' is being found first, then the 'opencalculator.api.OpenCalculateSatts.getInersteParentes()', 'opencalculator.api.OpenCalculate.countUtanParentes()', and 'opencalculator.api.OpenCalculateSatts.delInersteParentes()' called in sequence inside the 'if compare equals' condition. One can assume that if certain condition is met then these methods are performing some operations of the parenthesis or the contents of these parentheses.
- The error messages found in the bytecode refers to the erroneous situations with the parenthesis.
- All the method calls of the class 'opencalculator.api.OpenCalculateSatts' implies that this method is performing operations of the object of 'OpenCalculateSatts' class and its field 'Elementen', but it is difficult to conclude about its exact functionality.

Confidence Rank: Low
Value Ranking: Low

9. public getVariable(int) returns: double
Assumption: This method seems to retrieving value of some variables.

Method Bytecode:

- The following method calls are made:
  - Java.lang.Double.doubleValue () returns: double

- **Other information:**
  - A 'getfield' instruction is found with the field 'opencalculator.api.OpenCalculator.Variables'

Conjecture and Discussion:

- This method is returning the value of the field 'Variables' and casting it to return its value as double data type.

Confidence Rank: High
Value Ranking: Low

10. public setVariable (int,java.lang.Double) returns: void
Assumption: This method might be assigning value to the field 'Variables'

Method Bytecode:

- The following method calls are found:
  - NONE
- **Other information:**
  - The field 'Variables' is accessed.

Conjecture and Discussion:

- It seems that this method is assigning a value to the field 'Variables'

Confidence Rank: High
Value Ranking: Low

Final Conclusion for the class 'OpenCalculate':

- The methods 'getFunctionValue()'and 'getOperationValue()' performs arithmetical, trigonometric, and the logical operations.
- The methods 'HittaMinustal()' , 'UtforFunktioner()', 'UTforOperator()' performs operations on the object of class 'OpenCalculateSatts' and its field 'Elementen'. It seems that these methods tries to find some functions and operators, but it is difficult make any solid conclusions for them.
- The methods 'count()', 'getVariable()', 'setVariable()' operates on the field 'Variables' of this class and the 'OpenCalculateSatts' class.
- Most of the methods of this class uses methods defined in the class 'OpenCalculateSatts' and accepts an object of this class as their parameters. This

implies that the 'OpenCalculate' needs the 'OpenCalculateSatts' class to accomplish its purpose.

- The 'countUtanParentes()' methods performs some operations on the parentheses, but it is difficult to judge what functions it performs.
- Most of this class's methods creates an instance of the 'OCSyntaxError' class and throws exception.
- From the above assessments, one can conclude that this class has performs some basic arithmetic, trigonometric, and logical operations. The instance creation of the class 'OCSynataxError' implies that it might be accessing some functions, variables, parentheses, and it displays syntax errors.

Confidence rank: Medium
Value Ranking: High

h) Class Name: OpenCalculateKomando

Assumption: The English meaning of the word 'Komando' is 'command. The name of this class suggests that this class might be handling command operations for the Open Calculator application

Total Fields: Three

1. AttSkicka java.lang.Object

Assumption: The English translation of the words 'Att' and 'Skicka' are 'to' and 'send' respectively. It is difficult to assume about its purpose from its name.

2. Variables java.lang.Object[]

Assumption: This variable might be storing an array of some objects

3. io opencalculator.api.ioAble

Assumption: The type of this field implies that its purpose is to handle some input-out operations.

Total Methods: Eight

1. DoFunc(string) returns: boolean

Assumption: The name and the signature of this methods implies that it might be running some function and if it is executed successfully then true is returned, otherwise false is returned.

Method Bytecode:

- The following method calls are found:
  - java.lang.String.indexOf(string) returns: int
  - opencalculator.api.OpenCalculateKomando.checkIfValid (string, int) returns: boolean
  - java.lang.String.length() returns: int

- java.lang.String.subString (int) returns: string
- opencalculator.api.funcRunner.funcRunner (string)
  returns: java.lang.Object

- **Other information:**
  - A string <func.> is found loaded on the stack.
  - A 'if not equals' loop comparing the integers is found before the 'opencalculator.api.OpenCalculateKomando.checkIfValid()' method is called.
  - The value of the 'opencalculate.api.OpenCalculateKomando.Variables' is accessed and some value is assigned to the 'opencalculate.api.OpenCalculateKomando.AttSkicka' field.

Conjecture and Discussion:

- A string <func.> is found with the function call 'String.indexOf()', this implies that it might be returning the index within the string parameter of the first occurrence of the string 'func.'. One can assume that this method might be trying to locate the name class name from string with the format of 'opencalculator.api.func.XXX', which might be passed as a parameter to this method.
- From the brief assessment of the 'opencalculator.api.OpenCalculateKomando.checkIfValid()' method, one can conclude that it is validating the name of some function via calling methods of the 'java.lang.String' class. The method call 'opencalculator.api.OpenCalculateKomando.checkIfValid()' implies that the method 'DuFunc()' seems to be validating some functions before it operates on that function.
- It accesses the value of the 'opencalculate.api.OpenCalculateKomando.Variables' field, performs the 'String.substring()'operation on some word and then calls 'opencalculator.api.funcRunner.funcRunner()'. These sequence of instructions and method calls implies that the method 'DoFunc()' is retrieving the variables and running some function. These variables might be being passes as parameters to this function.
- It is difficult to judge why the 'DoFunc()' method accesses the 'opencalculate.api.OpenCalculateKomando.AttSkicka' field at the end of the method code.
- The method might be returning 'true' on successful execution of its functionality and 'false' on the failure.

Confidence Rank: Medium
Value Ranking: High

2. DoProg (string) returns: boolean

Assumption: This method seems to be performing some operation on some type of programs.

Method Bytecode:

- Following method calls are made:
  - java.lang.String.indexOf(string) returns: int
  - opencalculator.api.OpenCalculateKomando.checkIfValid(string,int) returns: boolean
  - java.lang.String.subString (int) returns: string
  - java.lang.StringBuilder.append (string) returns: java.lang.StringBuilder
  - opencalculator.api.progRunner.run (java.io.File) returns: java.lang.Object

- **Other information**
  - Following string are found in the bytecode:
    - "prog("
    - "opencalculator/program/"
    - "("
    - ")"
    - ".prg"
  - A string 'Programet exesterar inte' is found. Its English translation is 'Program UNKNOWN not'. This string is found along with the object instantiation of the class 'OCSyntaxError' inside a 'if not equals' condition.
  - Following fields are accessed:
    - opencalculator.api.OpenCalculate.Komando.io
    - opencalculate.api.OpenCalculateKomando.Variables
    - opencalculate.api.OpenCalculateKomando.Attskicka

Conjecture and Discussion:

- The method 'DoProg()' is finding the first occurrence of the index of a string 'prog(' within the string which is passed as an argument.
- The next method calls 'opencalculator.api.OpenCalculateKomando.checkIfValid ()' might be validating the format.
- The next set of 'String.indexOf()' and 'String.substring()' method calls, and the 'StringBuilder.append()' method calls implies that the name of the program is being retrieved via some string manipulations.
- The string declaration 'opencalculator/program' and the method call 'java.io.File.exists()' suggests that the 'DoProg()' method might be searching for some file in a particular directory.
- The method 'opencalculator.api.progRunner.run()' accepts a parameter of type 'java.io.File', this implies that the method 'DoProg()' is calling this method and passes some file name containing the program to be run.
- The method accesses 'opencalculator.api.OpenCalculate.Komando.io', 'opencalculate.api.OpenCalculateKomando.Variables', and 'opencalculate.api.OpenCalculateKomando.Attskicka', but it is difficult to judge the reasons why these fields are accessed here.

- From the above assessment and discussion, one can conclude that the method 'DoProg()' uses 'opencalculator.api.progRunner.run()' to run some program which is located inside a particular directory.

Confidence Rank: Medium
Value Ranking: High

3. DoString (string) returns: boolean

Assumption: This method seems to be performing certain operations on some string.

Method Bytecode:

- Following method calls are made:
  - java.lang.String.length () returns: int
  - java.lang.String.charAt (int) returns: char
  - java.lang.String.lastIndexOf (string) returns: int
  - java.lang.String.substring (int,int) returns: string

- **Other information**
  - The 'String.chartAt()' function is called within a loop.
  - A string 'Ingen giltig str?ng' [sic]' is found within a 'if compare not equals' condition along with an object instantiation of the 'OCSyntaxError' class. The English translation of this string may be 'Not valid string'
  - Two 'if' conditions are followed after the above code. A string '0till?tet tecken efter str?ng' [sic] is found inside one of the 'if' condition. Its English translation is '0 at token/char after string'.
  - A value is assigned to the 'opencalculate.api.OpenCalculateKomando.Attskicka' field.

Conjecture and Discussion:

- From the above observations made by exploiting the bytecode, one can assume that this method is performing some string manipulations and operations, and assigns some value to the 'opencalculate.api.OpenCalculateKomando.Attskicka'. Other then this, it is difficult to make exact conjecture about the purpose of this method.

Confidence Rank: Low
Value Ranking: Low

4. Dotilldelning (string) returns: boolean

Assumption: The English translation of the words 'till' and 'delning' is 'for/at/by/more' and 'parting'. It is difficult to assume about its functionality.

Method Bytecode:

- Following method calls are made:
  - java.lang.String.length () returns: int
  - java.lang.String.charAt (int) returns: char
  - java.lang.String.substring (int,int) returns: string
  - opencalculator.api.OpenCalculateKomando.run (string) returns: java.lang.Object

- **Other information**
  - The length of a string is calculated. This string seems to be the argument passed to the method.
  - The function 'String.charAt()'s is called inside of a loop. This implies that some characters are being searched for iteratively within a string.
  - A string 'Inget att tilldela variabeln' is found along with an object instantiation of 'OCSyntaxError' class. Its word-by-word English translation is 'not to allocation variables'.
  - Another loop found to have a long switch conditional branch. Each switch condition have an object instantiation of the 'OCSyntacError' class, a loop, and the following string declaration:
  - "F?r m?nga variabler att tilldela" [sic], its English translation is "UNKNOWN variable from/to allocate/assign".
  - Following two strings are found within 'if' condition:
    - "Det som skall tilldelas ?r ingen giltig variabel"[sic] , whose English translation is "this which/who/like must UNKNOWN none valid variable"
    - "Inget att tilldela variabeln".
  - The value of 'opencalculate.api.OpenCalculateKomando.Variables' field is accessed and some value is assigned to the 'opencalculate.api.OpenCalculateKomando.Attskicka' field.

Conjecture and Discussion:

- The switch condition seems to checking for some error conditions.
- This method is calling 'opencalculator.api.OpenCalculateKomando.run()' method, which seems to calling 'DoString()', 'DoFunc()', DoProg()', 'DoVariable()', and DoUtryck()'methods within the 'if…else' conditions.
- The bytecode assessment of this method, above information and discussion, does not provide any insights to conclude about its functionality.
- It is difficult to make any conjecture about this method's functionality.

Confidence Rank: Low
Value Ranking: Low

5. DoUtryck (string) returns: boolean

Assumption: The English translation of this method's name could not be found, so it is difficult to guess about its functionality.

Method Bytecode:

- Following method calls are made:
  - java.lang.String.length () returns: int
  - opencalculate.api.OpenCalculate.count (string) returns: double

- **Other information:**
  - A 'java.lang.Exception' is thrown within a 'if not equals' condition along with the string 'Ingenting att rakna'. Its English translation is 'Nothing that/to straighten'.
  - The value of 'opencalculate.api.OpenCalculateKomando.Variables' field is accessed and some value is assigned to the 'opencalculate.api.OpenCalculateKomando.Attskicka' field after the method call 'opencalculate.api.OpenCalculate.count()'.

Conjecture and Discussion:

- The exception is thrown after the length of the string parameter is found. The error message found along with the exception does not provide any information.
- It is difficult to make any conjecture about its functionality from the knowledge of its name, external method call 'opencalculate.api.OpenCalculate.count(), another bytecode other information.

Confidence Rank: Low
Value Ranking: Low

6. DoVariable (string) returns: boolean

Assumption: Its name implies that this method might be performing operations of some variables.

Method Bytecode:

- Following method calls are found:
  - java.lang.String.length () returns: int
  - java.lang.String.charAt (int) returns: char

- **Other information:**
  - A switch loop is found after the functions 'String.chartAt()' and 'String.length()' are called.
  - No significant instructions are found in the switch loop.
  - Two loops are found with no important instructions.
  - The value of 'opencalculate.api.OpenCalculateKomando.Variables' field is accessed and some value is assigned to the

'opencalculate.api.OpenCalculateKomando.Attskicka' field.

Conjecture and Discussion:

- The lack of significant information found inside the loops makes it difficult to assume about what is happen inside this method.
- From the name of this method and the discovery of some string functions, once can assume that this method might be performing some operations on some type of variables.

Confidence Rank: Low
Value Ranking: Low

7. checkIfValid (string, int) returns: boolean

Assumption: One can assume that this method might be performing some validations.

Method Bytecode:

- Following method calls are found:
  - java.lang.String.charAt (int) returns: char
  - java.lang.String.indexOf(string) returns: int
  - java.lang.StringBuffer.setCharAt(int, chat) returns: void
  - java.lang.String.length () returns: int

- **Other information:**
  - This method's bytecode has many 'if' conditions and some loops. The string operations are found inside these loops.
  - One 'if compare not equals' condition has an object instantiation of the 'OCSyntaxError' class and a string 'Ingen giltig funktion'. Its English translation is 'none valid function'.

Conjecture and Discussion:

- This method seems to validate some functions by performing string operations. These might be the functions necessary for Open Calculator application or some new functions which are added in this application's API package. But, still we don't have enough evidence to make any strong conclusions.

Confidence Rank: Low
Value Ranking: Low

8. run (string) boolean: java.lang.Object

Assumption: This method might be running some operations.

Method Bytecode:

- Following method calls are found:

- java.lang.String.length () returns: int
- opencalculator.api.OpenCalculateKomando.DoString (string)
  returns: boolean
- opencalculator.api.OpenCalculateKomando.Dotilldelning (string)
  returns: boolean
- opencalculator.api.OpenCalculateKomando.DoFunc (string)
  returns: boolean
- opencalculator.api.OpenCalculateKomando.DoProg(string)
  returns: boolean
- opencalculator.api.OpenCalculateKomando.DoVariable (string)
  returns: boolean
- opencalculator.api.OpenCalculateKomando.DoUtryck (string)
  returns: boolean

- **Other information:**
  - The value of 'opencalculate.api.OpenCalculateKomando.Attskicka' is accessed every time when each method with the word 'Do' in its name is called. These methods are called within 'if' conditions.
  - At the end of this method's bytecode, the field 'opencalculate.api.OpenCalculateKomando.Variable' is accessed.

Conjecture and Discussion:
- The primary function of this method is to call various local methods when certain condition is met.
- Nothing more can be assumed about what are these conditions and why the fields 'opencalculate.api.OpenCalculateKomando.Attskicka' and 'opencalculate.api.OpenCalculateKomando.Variables' are accessed.

Confidence Rank: Medium
Value Ranking: Medium

Final Conclusion about the class 'OpenCalculateKomando':
One can conclude this class has functionalities that perform the operations on some functions, program, variable, strings etc. All these functions have string operations, error messages, and the objects instantiations of the 'OCSynatxError' class. It seems this Open Calculator application might be accepting user commands from the command prompt, since this class seems to have many string operations.

Confidence Rank: Medium
Value Ranking: Low

i) Class Name: OpenCalculateSatts

Assumption: The English translation of the word 'satts' is unknown, so it is difficult to assume about its purpose.

Total Fields: Three
1. private Elementen java.util.Vector
Assumption: This field should be storing some elements.
2. private Parentes opencalculator.api.OpenCalculatorSatts$Parentes
Assumption: This field seems to represent the parenthesis.

3. private Variabler [sic] java.lang.Object[]
Assumption: It might be declared to represent some variables.

Total Methods: Fourteen
1. private HittaFunktion(int,string) returns: int

Assumption: The English meaning of the word 'Hitta' is 'find'. This method might be searching a function.

Method Bytecode:
- Following method calls are made:
  - java.lang.String.charAt (int) returns: char
  - java.lang.String.length () returns: int
  - java.util.Vector.addElementen(java.lang.Object) returns: void

- **Other information:**
  - String operations are found inside a switch loop.
  - The value of the field 'opencalculator.api.OpenCalculatorSatts.Elementen' is accessed at many places along with the 'java.util.Vector.addElementen()' function.

Conjecture and Discussion:
- This function adds a specified component at the end of the current Vector object.
- It seems that the 'HittaFunktion()' method performs string operations on the parameters passed to it, and adds new object in the 'Elementen' field.
- But one can not make any conclusions on what this 'Elementen' field is and what is the exact function of this method.

Confidence Rank: Low
Value Ranking: Low

2. private HittaOperator(int,string) returns: int
Assumption: This method might be searching for some operators.

Method Bytecode:
- Following method are called:

- java.lang.String.charAt (int) returns: char
- java.lang.String.length () returns: int
- java.util.Vector.addElementen(java.lang.Object) returns: void

- **Other information:**
  - A switch statement with many branches is found having loops, string operations, value of the field 'opencalculator.api.OpenCalculatorSatts.Elementen' is accessed, and the method 'java.util.Vector.addElement()' is called.

Conjecture and Discussion:
- The structure of this method's bytecode is similar to the method 'HittaFunktion()'.
- This implies that the method 'HittaOperator()' must be performing some operations on the field 'Elementen' for each operator, as the method 'HittaFunktion()' operates on some functions.

Confidence Rank: Medium
Value Ranking: Low

3. private HittaParentes(int,string) returns: int
Assumption: This method might be searching the parenthesis.

Method Bytecode:
- Following method calls are found:
  - java.lang.String.charAt (int) returns: char
  - java.lang.String.length () returns: int
  - java.util.Vector.addElementen(java.lang.Object) returns: void
  - opencalculator.api.OpenCalculatorSatts$Parentes.addVanster() returns: void
  - opencalculator.api.OpenCalculatorSatts$Parentes.addHoger() returns: void

- **Other information:**
  - The code has one switch loop along with many loops and 'if' conditions.
  - The value of the field 'opencalculator.api.OpenCalculatorSatts.Elementen' is accessed along with the method call 'java.util.Vector.addElementen()'.
  - The field 'opencalculator.api.OpenCalculatorSatts.Parenteser' is found being accessed along with the method calls 'addVanster()' and 'addHoger()'.

Conjecture and Discussion:
- The methods 'addVanster()' and 'addHoger()' are declared in the inner class 'Parentes' of the 'OpenCalculatorSatts' class and they are accessing the field 'opencalculator.api.OpenCalculatorSatts$Parentes.antal'. The meaning of the word 'antal' is 'count'. So, one can assume that the 'HittaParentes()' method is performing

some operations of the parentheses. These parentheses may be found in the command entered by the user. This parentheses does not seem to be part of any file since we did not find any 'java.io.File' objects.

Confidence Rank: Medium

Value Ranking: High

4. private HittaTal(int,string) returns: int

Assumption: The English meaning of the word 'tal' is 'number/sum'. Its name assume that this method might be searching for some numbers inside the string parameters.

Method Bytecode:

- Following method calls are made:
  - java.lang.String.charAt (int) returns: char
  - java.lang.String.length () returns: int
  - java.lang.StringBuilder.append(string) returns: java.lang.StringBuilder
  - java.util.Vector.addElementen(java.lang.Object) returns: void

- **Other information:**
  - A switch statement is found. Every branch of this switch statement has a loop along with the 'StringBuilder.append()' and 'String.charAt()' method calls.
  - The value of the 'opencalculator.api.OpenCalculatorSatts.Elementen' is accessed outside this switch statement.
  - A string with the message "Math Error: Inget gilltigt tal" is found. Its English translation is "Math Error: no UNKNOWN number/sum".

Conjecture and Discussion:

- The above observation implies that the method is performing only the string operations inside the switch loop and then accessing the 'opencalculator.api.OpenCalculatorSatts' and calling the 'java.util.Vector.addElementen()' method which adds a component at the end of the 'Elementen' field.
- The error message implies that this method is searching for some numbers.
- The above assessment is not enough to make any conclusion about this method's exact responsibility.

Confidence Rank: Low
Value Ranking: Low

5. private HittaVariabel(int,string)[sic] returns: int

Assumption: This method might be searching for some variables.

Method Bytecode:

- Following method are called:

- java.lang.String.charAt (int) returns: char
  - java.lang.String.length () returns: int
  - java.util.Vector.addElementen(java.lang.Object) returns: void
  - opencalculator.api.OpenCalculatorSatts.getVariable(int) returns: double

- **Other information:**
  - A big switch statement is found.
  - Each condition of the switch statement accesses 'opencalculator.api.OpenCalculatorSatts.Elementen' field and then calls 'opencalculator.api.OpenCalculatorSatts.getVariable()' and 'java.util.Vector.addElementen()'methods.

Conjecture and Discussion:

- The method 'opencalculator.api.OpenCalculatorSatts.getVariable()' returns the value of the field 'opencalculator.api.OpenCalculatorSatts.Variabler' [sic].
- From the bytecode assessment, one can conclude that this method access the values of 'opencalculator.api.OpenCalculatorSatts.Variabler'[sic] and 'opencalculator.api.OpenCalculatorSatts.Elementen' fields, and finally adds a component on the current 'Elementen' field.
- This method seems to be operating on the variables. These might be the variables which user passes as an argument while operating the Open Calculator or any other variables.

Confidence Rank: Medium
Value Ranking: Low

6. public delInersteParentes(java.lang.Double) returns: void

Assumption: The English meaning of the word 'Inerste' could not be found, so it is difficult to assume its functionality.

Method Bytecode:

- Following methods are called:
  - opencalculator.api.OpenCalculatorSatts$Parentes.getInersta() returns: int
  - java.util.Vector.elementAt(int) returns: java.lang.Object
  - java.lang.Integer.intValue() returns: int
  - java.util.Vector.remove(int) returns: java.lang.Object
  - java.util.Vector.set(int,java.lang.Object) returns: java.lang.Object

  - opencalculator.api.OpenCalculatorSatts$Parentes.deleteInersta() returns: void

- **Other information:**
  - The field ‘opencalculator.api.OpenCalculatorSatts.Elementen’ is accessed and the methods ‘java.util.Vector.elementAt()’ and ‘java.lang.Integer.intValue()’ are called inside a loop.
  - The value of the field ‘opencalculator.api.OpenCalculatorSatts.Parenteser’ is accessed along with the method call ‘opencalculator.api.OpenCalculatorSatts$Parentes.deleteInersta()’
  - The field ‘opencalculator.api.OpenCalculatorSatts.Elementen’ is again accessed when the methods ‘java.util.Vector.remove()’ and ‘java.util.Vector.set()’ are called.

Conjecture and Discussion:

- This method seems to be operating on the parentheses as the word ‘Parentes’ is found in its name.
- The field ‘opencalculator.api.OpenCalculatorSatts.Parenteser’ is of type ‘opencalculator.api.OpenCalculatorSatts$Parentes’ and this class also has a field ‘opencalculator.api.OpenCalculatorSatts$Parentes.parenteser’ of type array of boolean. One can assume from these information, that the method ‘delInersteParente()’ might be deleting or operating on pair of parentheses.
- The presence of method calls ‘java.util.Vector.remove()’, ‘java.util.Vector.set()’ implies that the value of the field ‘Elementen’ is manipulated.
- Even with all the above information, it is difficult to judge the exact functionality of this method.

Confidence Rank: Low
Value Ranking: Low

7. public getIdentifier(int) returns: int

Assumption: This method might be retrieving values of some type of identifier.

Method Bytecode:

- Following method calls are made:
  - java.uti.Vector.size() returns: int
  - java.util.Vector.elementAt(int) returns: java.lang.Object
  - java.lang.Integer.intValue() returns: int

- **Other information:**
  - The value of the field ‘opencalculator.api.OpenCalculatorSatts.Elementen’ is accessed and then the its size if calculated by ‘java.uti.Vector.size()’ function.
  - It is followed by a ‘multiplication’, ‘subtraction’ operations, and ‘if’ conditions.

- The field ‘opencalculator.api.OpenCalculatorSatts.Elementen’ is accessed again along with the method calls ‘java.util.Vector.elementAt()’ and ‘java.lang.Integer.intValue()’.

Conjecture and Discussion:

- It seems that within each ‘if’ condition a Vector component is accessed from the ‘Elementen’ field and arithmetical operations are performed on its Integer value, which is obtained by the casting function ‘java.lang.Integer.intValue()’.
- But, it is difficult to make any conjecture that what are these identifies that this method is retrieving

Confidence Rank: Low
Value Ranking: Low

8. public getInersteParentes()
return: opencalculator.api.OpenCalculatorSatts

Assumption: One can assume that this method might be retrieving some values inside the parentheses.

Method Bytecode:

- Following methods are called:
  - opencalculator.api.OpenCalculatorSatts$Parentes.getInersta() returns: int
  - java.util.Vector.elementAt(int) returns: java.lang.Object
  - java.lang.Integer.intValue() returns: int
  - java.util.Vector.add(java.lang.Object) returns: void

- **Other information:**
  - The field ‘opencalculator.api.OpenCalculatorSatts.Prenteser’ is accessed before the method ‘opencalculator.api.OpenCalculatorSatts$Parentes.getInersta()’ is called.
  - It is followed by couple of loops in which the field ‘opencalculator.api.OpenCalculatorSatts.Elementen’ is accessed and the methods ‘java.util.Vector.elementAt()’ and ‘java.lang.Integer.intValue()’ are called.
  - The next set of loops has method call ‘java.util.Vector.add()’ along with the accessing the value of the field ‘Elementen’.
  - Before the final return statement, the field ‘opencalculator.api.OpenCalculatorSatts.Variabler’ is accessed.

Conjecture and Discussion:

- One can not conclude about the exact functionality of this method, but it seems to be performing following:

- This method is calling its inner class method 'opencalculator.api.OpenCalculatorSatts$Parentes.getInersta()', which seems to perform some operations on the parentheses.
- It accesses the components from the 'Elementen' field by calling the 'java.util.Vector.elementAt()' method, it adds more components and returns the object of 'opencalculator.api.OpenCalculatorSatts' class.

Confidence Rank: Low
Value Ranking: Low

9. public getValue(int) returns: double
Assumption: This method seems to be retrieving some value.

Method Bytecode:
- Following methods are called:
  - java.util.Vector.elementAt(int) returns: java.lang.Object
  - java.lang.Double.doubleValue() returns: double

- **Other information:**
  - The value of the field 'opencalculator.api.OpenCalculatorSatts.Elementen' is accessed.

Conjecture and Discussion:
- This method accessed a particular Vector component from the field 'opencalculator.api.OpenCalculatorSatts.Elementen' and then returns it by casting it using 'java.lang.Double.doubleValue()' function.

Confidence Rank: Medium
Value Ranking: Low

10. public getVariable(int) returns: double
Assumption: This method might be returning the value of the field 'opencalculator.api.OpenCalculatorSatts.Variabler'[sic].

Method Bytecode:
- Following methods are called:
  - java.lang.Double.doubleValue() returns: double

- **Other information:**
  - The value of the field 'opencalculator.api.OpenCalculatorSatts.Variabler'[sic] is accessed.
  - A 'if' condition is found a return statement and the 'java.lang.Double.doubleValue()' function call.
  - A text string is found 'Variabel [sic] g?r [sic] inte att r?kna med'. Its

word-by-word English translation is ‘variable UNKNOWN not that/to UNKNOWN with/by’
- An object is instantiated of the ‘OCVariableError’ class along with the above error message.
-

Conjecture and Discussion:
- One can conclude that this method accesses the ‘opencalculator.api.OpenCalculatorSatts.Variabler’[sic] field, casts it to ‘double’ type, and if the certain condition is met then returns it.
  It instantiates ‘OCVariableError’ when the condition is not met for the field ‘opencalculator.api.OpenCalculatorSatts.Variabler’[sic].

Confidence Rank: Medium
Value Ranking: Low

11. public remove(int) returns: void
Assumption: This method seems to be removing something, which might be an integer value.

Method Bytecode:
- Following method calls are made:
  - java.uti.Vector.remove(int) returns: java.lang.Object

- **Other information:**
  - The code length is 23.
  - The field ‘opencalculator.api.OpenCalculatorSatts.Elementen’ is accessed.

Conjecture and Discussion:
- This method seems to be removing a Vector component from the field ‘Elementen’

Confidence Rank: High
Value Ranking: Low

12. public setNumber(int,double) returns: void
Assumption: This method seems to be setting the values of some number.

Method Bytecode:
- Following methods are called:
  - java.util.Vector.set(int,java.lang,Object) returns: java.lang.Object

- **Other information:**

- The field ‘opencalculator.api.OpenCalculatorSatts.Elementen’ is accessed.
- The class ‘java.lang.Integer’ is initiated after a ‘multiply’ instruction.
- The class ‘java.lang.Double’ is initiated after a pair of ‘multiply’ and ‘add’ instructions.

Conjecture and Discussion:

- The presence of the java.util.Vector.set() function and the above observation implies that this method might be swapping the components by changing the index of the vector field ‘opencalculator.api.OpenCalculatorSatts.Elementen’.

Confidence Rank: Medium
Value Ranking: Low

13. public setVariable(int,java.lang.Double) returns: void
Assumption: This might be a setter method that assigns a value to the field ‘opencalculator.api.OpenCalculatorSatts.Variabler’[sic]

Method Bytecode:

- Following methods are called:
  - NONE
- **Other information:**
  - The field ‘opencalculator.api.OpenCalculatorSatts.Variabler’[sic] is accessed.

Conjecture and Discussion: This method assigns a value to the field ‘opencalculator.api.OpenCalculatorSatts.Variabler’[sic].

Confidence Rank: High
Value Ranking: Low

Final Conclusion for the class ‘OpenCalculateSatts’:

- The methods like ‘HittaTal()’, ‘HittaVariabel()’[sic], ‘HittaOperator()’, ‘HittaFunktion()’, ‘HittaParentes()’ performs operation on some numbers, variables, operators, functions, and parentheses respectively. It seems that these are found in the user command that might have been entered at the command prompt interface.
- The other methods of this class seems to perform operations on the parentheses.

Confidence Rank: Medium
Value Ranking: Low

i) Class Name: OpenCalculateSatts$1

Assumption: This is an anonymous inner class of the 'OpenCalculateSatt' class.

Final Conclusion: Nothing can be concluded about its purpose and functionality since no information is found for this class.

j) Class Name: OpenCalculateSatt$Parentes

Assumption: The meaning of the word 'Parentes' is 'parentheses'. One assume that this inner class should have some functionalities that operate on the parentheses.

Total Fields: Three

1. private antal int

Assumption: The English meaning of the word 'antal' is 'count'. So, one can assume that this variable might be storing the parentheses count.

2. private parenteser boolean[]

Assumption: The English translation of the word 'parenteser' is 'parentheses'. But, as its data type is 'boolean[]', it is difficult to assume the purpose of this variable.

3. final this$0 opencalculator.api.OpenCalculatorSatts

Assumption: Its name seems to be obfuscated, so it is difficult to assume about its functionality.

Total Methods: Five

1. public addHoger() returns: void

Assumption: The English meaning of the word 'Hoger' could not be found, so one can not assume about this methods functionality.

Method Bytecode:

- Following method calls are found:
  - NONE
- **Other information:**
  - The code length is 21.
  - The value of the field 'opencalculator.api.OpenCalculatorSatts$Parentes.parenteser' is accessed.
  - Some value is assigned to the field 'opencalculator.api.OpenCalculatorSatts$ Parentes.antal'

Conjecture and Discussion:

- Based on the value of the field 'opencalculator.api.OpenCalculatorSatts$Parentes. parenteser', some value is assigned to the field 'opencalculator.api.OpenCalculatorSatts$ Parentes.antal' after a 'addition' instruction.
- It is difficult to conclude this method's purpose.

Confidence Rank: Medium

Value Ranking: Low

2. public addVanster() returns: void

Assumption: The English translation of the word ‘vanster’ could not be found, so this method’s functionality can not be guessed.

Method Bytecode:

- Following methods are called:
  - NONE
- **Other information:**
  - The code length is 21.
  - The value of the field ‘opencalculator.api.OpenCalculatorSatts$Parentes.parenteser’ is accessed.
  - Some value is assigned to the field ‘opencalculator.api.OpenCalculatorSatts$ Parentes.antal’

Conjecture and Discussion:

- The bytecode structure of this method and the ‘addHoger()’ is almost similar.
- Since the meaning of the word ‘Hoger’ is unknown, it is difficult to conjecture about this method’s functionality, except one can say that it assigns some value to the ‘opencalculator.api.OpenCalculatorSatts$ Parentes.antal’ field.

Confidence Rank: Medium
Value Ranking: Low

3. public deleteInersta() void

Assumption: The English meaning of the word ‘Inersta’ is unknown, so it is difficult to assume about this method’s functionality except it should be performing some delete operations

Method Bytecode:

- Following method calls are made:
  - NONE
- **Other information:**
  - The fields ‘opencalculator.api.OpenCalculatorSatts$Parentes.parenteser’ and ‘opencalculator.api.OpenCalculatorSatts$Parentes.antal’ is accessed many time in the code within loops.
  - Following two string message were declared:
    - “En parentes ?r felaktig”. Its English translation is “one parenthesis UNKNOWN incorrect”.
    - “Ingen Parantes”. Its English translation is ‘no parentheses’.

- An object is instantiated for “OCSyntaxError” class along with the above error messages.

Conjecture and Discussion:

- The fields ‘opencalculator.api.OpenCalculatorSatts$Parentes.parenteser’ and ‘opencalculator.api.OpenCalculatorSatts$ Parentes.antal’ are accessed inside loops. It implies that their values are accessed iteratively.
- The error message implies that this method is performing delete operations on parentheses. These parentheses are might be the part of the command given or entered by the user.

Confidence Rank: Medium
Value Ranking: Low

4. public getAntal() returns: int

Assumption: This method seems to retrieving the value of the ‘antal’ field.

Method Bytecode:

- NONE
- **Other information:**
  - The code length is 5.
  - The field ‘opencalculator.api.OpenCalculatorSatts$ Parentes.antal’ is being accessed

Conjecture and Discussion:

- This method retrieves the value of the field ‘opencalculator.api.OpenCalculatorSatts$ Parentes.antal’.

Confidence Rank: High

Value Ranking: Low

5. public getInersta() returns: int

Assumption: The English meaning of the word ‘inersta’ is not found. It is difficult to assume about this method’s functionality.

Method Bytecode:

- Following method calls are found:
  - NONE
- **Other information:**
  - The fields ‘opencalculator.api.OpenCalculatorSatts$Parentes.parenteser’ and ‘opencalculator.api.OpenCalculatorSatts$ Parentes.antal’ is accessed many time in the code within loops.
  - Following two string message were declared:

- "En parentes ?r felaktig". Its English translation is "one parenthesis UNKNOWN incorrect".
- "Ingen Parantes". Its English translation is "no parentheses".

- An object is instantiated for 'OCSyntaxError' class along with the above error messages.

Conjecture and Discussion:

- The fields 'opencalculator.api.OpenCalculatorSatts$Parentes.parenteser' and 'opencalculator.api.OpenCalculatorSatts$Parentes.antal' are accessed inside loops. It implies that their values are accessed iteratively.
- The error message implies that this method is performing retrieving operations on parentheses. These parentheses are might be the part of the command given or entered by the user.

Confidence Rank: Medium
Value Ranking: Low

Final Conclusion for the class 'OpenCalculateSatt$Parentes':

- This is the inner class of the 'OpenCalculateSatt', its name suggests and the above assessment does not provide enough information to conclude about its purpose and functionality. All the methods declared in this class access the fields 'opencalculator.api.OpenCalculatorSatts$Parentes.parenteser' and 'opencalculator.api.OpenCalculatorSatts$Parentes.antal'. Noting more can be concluded except this class seems to operate on the parentheses.

Confidence Rank: Low
Value Ranking: Low

k) Class Name: program

Assumption: One can assume that this class's purpose seems to control programs. But, it is difficult to conclude what these programs are.

Total Fields: NONE

Total Methods: One

1. public run() returns: opencalculator.api.programReturn

Assumption: This method seems to run the programs.

Method Bytecode:

- Following methods calls are made:
  - NONE
- **Other information:**
  - Code length is 8.
  - An object of the class 'opencalculator.api.programReturn' is invoked.

Conjecture and Discussion:
- The bytecode of this method has only one significant instruction 'opencalculator.api.programReturn<init>>'. Nothing can be concluded about its functionality except, it is dependent the class 'programReturn'.

Confidence Rank: Low
Value Ranking: Low

Final Conclusion for the class 'program':
- This class has only one method declared in it and no field is declared. The assessment of its method 'run()' didn't provided any clues about its functionalities. Nothing significant can be concluded about its purpose.

Confidence Rank: Low

Value Ranking: Low

l) Class Name: ***programKomando***
Assumption: It is an abstract class. This class might be defining some types of command operations for some programs. It

Total Fields: One

1. *private* *type* *int*
Assumption: This field might be assigning numbers to some kind of program types.

Total Methods: One

1. public getType() int
Assumption: This method should be returning the value of the field 'type'.

Method Bytecode:
- Method calls: None
- **Other information:**
  - The field 'opencalculator.api.programKomando.type' is accessed.

Conjecture and Discussion: It returns the value of the field 'opencalculator.api.programKomando.type'.

Confidence Rank: High
Value Ranking: Low

Final Conclusion for the class 'programKomando'-
- It is an abstract class and it provide a method definition that returns the program type.

Confidence Rank: High

Value Ranking: Low

m) Class Name: **programList**

Assumption: One can assume that this class may be handling the program lists.

Total Fields: Five

1. private forstaLines boolean

Assumption: The English translation of 'forsta' or 'sta' is unknown. One can assume that this field might represent some types of lines.

2. komandoList java.util.Vector

Assumption: This field might be representing the list of commands.

3. nSetLine int

Assumption: This field might be representing the number of lines which are set.

4. private nextCounter int

Assumption: This field might be representing the next number of some type of counter .

5. private radnummer java.util.Vector

Assumption: The English meaning of 'radnummer' is 'line number'. This field seems to represent some kind of line number.

Total Methods: Fifteen

1. private checkLast(opencalculator.api.programKomando) returns: int

Assumption: This method might be checking the last type of some program command.

Method Bytecode:

- Following methods are called:
  - opencalculator.api.vilkorsProgramsKomando.getKomando() returns: opencalculator.api.programKomando
  - opencalculator.api.vilkorsProgramsKomando.whitElse() returns: boolean
  - opencalculator.api.vilkorsProgramsKomando.getElseKomando() returns: opencalculator.api.programKomando
  - opencalculator.api.whileProgramsKomando.getKomando() returns: opencalculator.api.programKomando

- **Other information:**
  - All the above methods are called within a ‘if equals’ condition along with an instance creation of the class ‘opencalculator.api.programList’
  - A ‘checkcast’ instruction is found with ‘opencalculator.api.vilkorsProgramsKomando’ and the ‘opencalculator.api.whileProgramsKomando’ within then above ‘if’ conditions.

Conjecture and Discussion:
- Too many external method calls make it difficult an extensive assess the bytecode.
- The above assessment implies that this method is calling different methods of the class ‘opencalculator.api.vilkorsProgramsKomando’ and ‘opencalculator.api.whileProgramsKomando’ when certain conditions are met and then creates an instance of the class ‘opencalculator.api.programList’.

Confidence Rank: Low
Value Ranking: Low

2. public clearCounter() returns: void
Assumption: This method seems clearing the current value of some counter.

Method Bytecode:
- Following methods are called:
  - NONE
- **Other information:**
  - The field ‘opencalculator.api.programList.nextCounter’ is assigned a value.

Conjecture and Discussion:
-It seems that this method clears the current value of some counter and assigns it to the field ‘nextCounter’.

Confidence Rank: Medium
Value Ranking: Low

3. private doTokenizon(string) returns: java.util.Vector
Assumption: This method seems tokenizing the string argument and creates a Vector instance.

Method Bytecode:
- Following methods are called:
  - java.lang.String.indexOf(int) returns: int

- java.lnag.String.length() returns: int
- java.lang.String.charAt(int) returns: char
- opencalculator.api.programList.doTokenizon(string) returns: java.util.Vector
- java.util.Vector.addElement(java.lang.Object) returns: void
- java.util.Vector.removeElement(java.lang.Object) returns: void
- java.util.Vector.elementAt(int) returns: java.lang.Object
- java.util.Vector.size() returns: int
- java.util.Vector.lastElement() returns: java.lang.Object
- java.lang.String.lastIndexOf(int) returns: int
- java.lang.StringTokenizer.countTokens() returns: int

- **Other information:**
  - The above string operation along with some loops seems to be tokenizing the string.
  - A sting 'Prog err inget slut p? {parantes' is found. Its English translation is 'Program error not end/finish UNKNOWN { paranthesis'.
  - An object of the class 'OCPError' is instantiated.
  - A string constant ";?2#a;" is found along with the function call 'StringBuffer.replace()'.
  - A string constant "?2#a" is found along with the function call 'String.equals()'.
  - A string constant ";" is found along with the function call 'StringTokenizer.countToken()'.
  - An error message 'program err. ; Excepter at last line'. Its English translation is 'program error: ; except at last line'.

Conjecture and Discussion:
- It seems that this method perform the string operations. The string tokenizing function found with the string constant ";" implies that ";" might be the delimiter for the string tokenizer. The presence of the string ";?2#a;" does not provide any clue about its presence.
- The error message suggests that this method is searching for "}" and ";" characters.
- The exact functionalities of this method could not be concluded.

Confidence Rank: Medium
Value Ranking: Low

4. private getKomandoList(java.util.Vector) returns: java.util.Vector

Assumption: This method might be retrieving the list of some commands.

Method Bytecode:

- Following methods are called:
  - java.util.Vector.size() returns: int
  - opencalculator.api.programList.makeKomando(java.util.Vector, int[]) returns: opencalculator.api.whileProgramsKomando'
  - java.util.Vector.addElement(java.lang.Object) returns: void

- **Other information:**
  - **A** new array of integers is created.
  - The methods 'opencalculator.api.programList.makeKomando()' and 'java.util.Vector.addElement()' are called within a loop.

Conjecture and Discussion:
- It is not understood that why the array of integers is being created.
- The method calls 'opencalculator.api.programList.makeKomando()' and 'java.util.Vector.addElement()' within a loop implies that the method 'getKomandoList()' is calling 'makeKomando()' to add elements in the Vector component.
- From the above assessment and discussion, it is difficult to make any conjectures about this method's functionality.

Confidence Rank: Low
Value Ranking: Low

5. private getKomandoStrings(java.io.LineNumberReader,int) returns: java.util.Vector

Assumption: This method seems to be retrieving command strings.

Method Bytecode:
- Following methods are called:
  - java.io.LineNumberReader.readLine() returns: string
  - java.lang.String.length() returns: int
  - java.lang.String.indexOf(string) returns: int
  - java.lang.StringBuffer.delete(int,int) returns: java.lang.StringBuffer
  - opencalculator.api.programList.doTokenizon(string) returns: java.util.Vector

- **Other information:**
  - A string "//" is found with the function 'String.indexOf()'.
  - The fields 'opencalculator.api.programList.forstaLines' and 'opencalculator.api.programList.radnummer' are accessed after all the string operations are done.

Conjecture and Discussion:
- From the byte code assessment and above discussion one can conclude that these methods seems to be reading each line and then performing

string tokenizing function.
- It accesses the field ‘opencalculator.api.programList.radnummer’ and assigns some value to ‘opencalculator.api.programList.forstaLines’.

Confidence Rank: Medium
Value Ranking: Low

6. public getNext()
returns: opencalculator.api.programKommando

Assumption: This is method seems to retrieve the next command program

Method Bytecode:
- Following methods are called:
  - java.util.Vector.elementAt(int) returns: java.lang.Object

- **Other information:**
  - The value of the field ‘opencalculator.api.programList.nextCounter’ is accessed followed by an ‘add’ instruction.
  - The field ‘opencalculator.api.programList.komandoList’ is accessed.
  - A ‘subtract’ instruction is followed by the function call ‘java.util.Vector.elementAt()’.

Conjecture and Discussion:
- The bytecode assessment implies that the value of the field ‘nextCounter’ is updated by performing an addition operation of the original value of the ‘nextCounter’.
- It seems that the function call ‘Vector.elementAt()’ returns an object of the type ‘opencalculator.api.programKommando’ , which is returned by the ‘getNext()’ method.
- One can make a final conjecture about this method’s functionality that it updates the value of the field ‘nextCounter’, and then returns next component of the Vector ‘komandoList’.

Confidence Rank: Medium
Value Ranking: Low

7. private makeKomando(java.util.Vector, int[])
returns: opencalculator.api.programKomando

Assumption: This method might be creating new type of commands.

Method Bytecode:
- Following methods are called:
  - java.util.Vector.elementAt(int) returns: java.lang.Object
  - opencalculator.api.programList.vilkorsKommando(string)

returns: boolean

- opencalculator.api.programList.makeVilkorsKomando(java.util.Vector,int[])
returns: opencalculator.api.vilkorsProgramKomando
- opencalculator.api.programList.checkLast(opencalculator.api.programKomando) returns: int
- opencalculator.api.programList.whileKommando(string)
returns: int
- opencalculator.api.programList.makeWhileKomando(java.util.Vector,int[])
returns: opencalculator.api.whileProgramKomando

Other information:

- A 'checkcast' instruction is found with an object instantiation of 'java.lang.String' class. This is followed by the method call 'opencalculator.api.programList.vilkorsKommando()'.
- It is followed by a 'if equals' condition containing the method calls 'opencalculator.api.programList.makeVilkorsKomando()' and 'java.util.Vector.elementAt()'.
- The method calls 'opencalculator.api.programList.whileKommando()' and 'opencalculator.api.programList.makeWhileKomando()' have the same code structure as above.
-

Conjecture and Discussion:

- This method calls other methods to accomplish its responsibility. One can not make any conjectures about this method's exact functionality, because it is dependent on may other methods.
- Based upon the assessment, it seems that it is creating different command types.

Confidence Rank: Low
Value Ranking: Low

Final conclusion about the class 'programList':

- Based on the above assessment of some member methods, one can make following conjectures about this class:
  - This class has methods which operate on some 'programs', its variables, code, and operators.
  - The string manipulation functions suggest that this class is operating on some type of code. Presence of some of the methods implies that this class controls some types of commands.
  - This application has different components to control 'functions' and 'programs'. Thus, one can assume that the 'functions' and 'programs' has different responsibilities for this application. The class 'programList' has responsibilities to control some 'programs' for this application.

Confidence Rank: Medium

Value Ranking: High

Level 0 Assessment based on the hierarchies and relationship amongst the classes found under the 'API' package:

Hierarchies: Following inheritance relationships are found in this package:

1. The classes 'OCPError', 'OCSyntaxError', 'OCVariableError', and 'OCprogramError' are subclasses of 'OCError' class.
Conjecture: The classes which are inherited from the class 'OCError' represents different types of error that can occur in the Open Calculator application.

Confidence Rank: High

2. The classes 'programList', 'vanligtProgramKomando', 'vilkorsProgramKomando', 'whileProgramKomando' are inherited from the class 'programKomando'.
Conjecture: The subclasses of the class 'programKomando' seem to represent different type of command programs. Nothing could be concluded about these program types.

Confidence Rank: Low

Level 1 Assessment for the API package.

Package Name: API
Assumption: Application Programming Interface (API) contains a set of components that is used by the other packages of the application or the other software applications. One can assume that it must be providing functionalities that are used by other packages of the Open Calculator application.

Final Conclusion: This package has classes are defined the errors, programs, functions, and the classes that run the functions and the programs.

Confidence Rank: High

Value Ranking: High

High Level Assessment of the 'Interface' package:
The vulnerability assessment of the 'Interface' package is done based on the UML diagram developed using jGRASP. The appendix A has the UML diagram and the Appendix B has the UML documentations.

- ‘OCMain’, ‘graph’, ‘graphpainter’, ‘program’, ‘programEditor’, and ‘console’ are some of the classes of interest.
- The member fields and the member methods of the classes ‘graph’ and ‘graphpainter’ suggest that the Open Calculator has graphing utility.
- Most of the classes have fields declared of the type ‘javax.swing.JButton’, ‘javax.swing.JPanel’, ‘javax.swing.JTextField’. This implies that this application should have a graphical user interface.
- The presence of ‘java.awt.event.ActionEvent’, ‘java.awt.event.WindowEvent’, ‘keyPressed() returns: java.awt.event.KeyEvent’, etc implies that this application has interactive graphical user interface.
- One can make a conjecture that this package handles the interface of the Open Calculator.

Confidence Rank: Medium

Value Rank: High

Level 3 Assessment for the Open Calculator application:

- The vulnerability assessment of the packages and the classes of this application help us to make some conjectures about its overall functionality.
- As the name of the application is ‘Open Calculator’, one can assume that this should be an open source calculator application and user should be able to customize it.
- The assessment done on the classes ‘OCSyntaxError’, ‘OCprogramError’, ‘OCVariableError’, ,’funcRunner’, ‘programRunner’ suggests that this application does operations such as, reads some file, parses it, searches for some programs and functions, displays error and line numbers where the errors have occurred, etc. One can make conjecture that the user might be able to add their own function definitions or programs to customize the application.
- It is difficult to make any final conclusions about its user interface. The assessment of the ‘Interface’ package implies that it has a graphical user interface, whereas some of the evidence found during the ‘API’ assessment implies that it has a command line interface.

Confidence Rank: Medium

Value Ranking: High

APPENDIX D

I. Method Name: runFunc()

Class Name: funcRunner

Confidence Rank: Medium

a) Byte Code

```
 0 aload_1
 1 iconst_0
 2 aload_1
 3 bipush 40
 5 invokevirtual #18 <java/lang/String.indexOf>
 8 invokevirtual #19 <java/lang/String.substring>
11 astore_2
12 aload_0
13 new #5 <java/lang/StringBuilder>
16 dup
17 invokespecial #6 <java/lang/StringBuilder.<init>>
20 ldc #20 <opencalculator.api.func.>
22 invokevirtual #7 <java/lang/StringBuilder.append>
25 aload_2
26 invokevirtual #7 <java/lang/StringBuilder.append>
29 invokevirtual #9 <java/lang/StringBuilder.toString>
32 invokevirtual #21 <opencalculator/api/funcRunner.findClass>
35 astore_3
36 aload_0
37 aload_1
38 aload_1
39 bipush 40
41 invokevirtual #18 <java/lang/String.indexOf>
44 iconst_1
45 iadd
46 aload_1
47 bipush 41
49 invokevirtual #22 <java/lang/String.lastIndexOf>
52 invokevirtual #19 <java/lang/String.substring>
55 invokespecial #23 <opencalculator/api/funcRunner.getParameters>
58 astore 4
60 aload_0
61 aload 4
63 invokespecial #24 <opencalculator/api/funcRunner.getTypes>
66 astore 5
68 aload_3
69 new #5 <java/lang/StringBuilder>
```

```
 72 dup
 73 invokespecial #6 <java/lang/StringBuilder.<init>>
 76 ldc #25 <FUNC>
 78 invokevirtual #7 <java/lang/StringBuilder.append>
 81 aload_2
 82 invokevirtual #7 <java/lang/StringBuilder.append>
 85 invokevirtual #9 <java/lang/StringBuilder.toString>
 88 aload 5
 90 invokevirtual #26 <java/lang/Class.getMethod>
 93 aconst_null
 94 aload 4
 96 invokevirtual #27 <java/lang/reflect/Method.invoke>
 99 areturn
100 astore_2
101 new #15 <java/lang/Exception>
104 dup
105 ldc #28 <Funktionen hittades ej>
107 invokespecial #29 <java/lang/Exception.<init>>
110 athrow
111 astore_2
112 new #15 <java/lang/Exception>
115 dup
116 aload_2
117 invokevirtual #30 <java/lang/Exception.getMessage>
120 invokespecial #29 <java/lang/Exception.<init>>
123 athrow
```

b) Decompiled Code

```
// Decompiled by Jad v1.5.8f. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)
// Source File Name:   funcRunner.java

public Object runFunc(String s)
        throws Exception
    {
        String s1;
        Class class1;
        Object aobj[];
        Class aclass[];
        s1 = s.substring(0, s.indexOf('('));
        class1 = findClass((new StringBuilder()).
            append("opencalculator.api.func.").append(s1).toString());
        aobj = getParameters(s.substring(s.indexOf('(') + 1,
                                                 s.lastIndexOf(')')));
        aclass = getTypes(aobj);
        return class1.getMethod((new
            StringBuilder()).append("FUNC").append(s1).toString(),
                                             aclass).invoke(null, aobj);
        Object obj;
        obj;
        throw new Exception("Funktionen hittades ej");
        obj;
```

```
        throw new Exception(((Exception) (obj)).getMessage());}
```

c) Byte Code Assessment Conjectures and Discussion

public runFunc(string) returns: java.lang.Object
Assumption: The name of this method suggests that it may be running the function of the name passed to it as a parameter.

Method Bytecode:

- The function 'runFunc()' makes the following method calls:
  - java.lang.String.indexOf (int) returns: int
  - java.lang.String.substring (int,int) returns: string
  - java.lang.StringBuilder.append (string) returns: java.lang.StringBuilder
  - opencalculator.api.funcRunner.findClass (string) returns: java.lang.Class
  - java.lang.String.substring (int,int) returns: string
  - opencalculator.api.funcRunner.getParameters (string) returns: Objects []
  - opencalculator.api.funcRunner.getTypes (Objects[]) returns: Class []
  - java.lang.Class.getMethod (string,java.lang.Class[]) returns: java.lang.Method
  - java.lang.reflect.Method.invoke (Object, Object[]) returns: Object
  - java.lang.Exception.getMessage () returns: string

- **Other Information**:
  - Stack loads a string constant of value "opencalculator.api.func." followed by a 'StringBuilder.append()' function.
  - Another string constant if found of the value "FUNC".
  - A string declaration is found of the value "Funktionen hittades ej". It English translation is "Function UNKNOWN not".
  - It is followed by 'java.lang.Exception.<init>>' and a 'athrow' instruction.

Discussion and Conjecture:

- The method 'runFunc()' performs some string operations on the parameters passed to it and then a string "opencalculator.api.func." is appended to it and tries to build a string like "opencalculator.api.func.xxxxxx". This implies that the method is accessing classes from the "opencalcualtor.api.func" package.
- The string constant "FUNC" is appended to another string. Interestingly the word "FUNC" is found in all the methods declared in the 'api.func' package.
- Then the method call 'funcRunner.findClass()' is searching for that particular class and reads its contents byte by byte.
- The method calls 'funcRunner.getParameters()' and 'funcRunner.getTypes()' suggests that the parameters and their types are being accessed from that class or the class file in the 'opencalculator.api.func' directory.
- The function 'java.lang.reflect.Method.invoke()' is called after the

‘java.lang.Class.getMethod()’ function is invoked. It is evident that the method with a string “FUNC” in its name and which is declared in “opencalculator.api.func.xxxxxxx’ class is retrieved and invoked using the ‘java.lang.reflect.Method.invoke()’ function.

- The exception message is displayed with the string “Function not UNKNOWN”. The word UNKNOWN is used since the English translation of the word ‘hittades is not found. This is a non-English string found in the bytecode along with the code dealing with the ‘java.lang.Exception’ class.

Confidence Rank: Medium
Value Ranking: High

II. Method Name: findClass()

Class Name: funcRunner

Confidence Rank: Medium

a) Byte Code

```
0 aconst_null
 1 astore_2
 2 aload_1
 3 bipush 46
 5 bipush 47
 7 invokevirtual #3 <java/lang/String.replace>
10 astore_3
11 new #4 <java/io/FileInputStream>
14 dup
15 new #5 <java/lang/StringBuilder>
18 dup
19 invokespecial #6 <java/lang/StringBuilder.<init>>
22 aload_3
23 invokevirtual #7 <java/lang/StringBuilder.append>
26 ldc #8 <.class>
28 invokevirtual #7 <java/lang/StringBuilder.append>
31 invokevirtual #9 <java/lang/StringBuilder.toString>
34 invokespecial #10 <java/io/FileInputStream.<init>>
37 astore_2
38 aload_2
39 invokevirtual #11 <java/io/FileInputStream.available>
42 newarray 8 (byte)
44 astore 4
46 aload_2
47 aload 4
49 invokevirtual #12 <java/io/FileInputStream.read>
52 pop
53 aload_0
```

```
 54 aload_1
 55 aload 4
 57 iconst_0
 58 aload 4
 60 arraylength
 61 invokevirtual #13 <opencalculator/api/funcRunner.defineClass>
 64 astore 5
 66 aconst_null
 67 aload_2
 68 if_acmpeq 80 (+12)
 71 aload_2
 72 invokevirtual #14 <java/io/FileInputStream.close>
 75 goto 80 (+5)
 78 astore 6
 80 aload 5
 82 areturn
 83 astore_3
 84 new #16 <java/lang/ClassNotFoundException>
 87 dup
 88 aload_1
 89 invokespecial #17 <java/lang/ClassNotFoundException.<init>>
 92 athrow
 93 astore 7
 95 aconst_null
 96 aload_2
 97 if_acmpeq 109 (+12)
100 aload_2
101 invokevirtual #14 <java/io/FileInputStream.close>
104 goto 109 (+5)
107 astore 8
109 aload 7
111 athrow
```

b) Decompiled Code

```
protected Class findClass(String s)
        throws ClassNotFoundException
    {
        FileInputStream fileinputstream = null;
        Class class1;
        try
        {
            String s1 = s.replace('.', '/');
            fileinputstream = new FileInputStream((new
StringBuilder()).append(s1).append(".class").toString());
            byte abyte0[] = new byte[fileinputstream.available()];
            fileinputstream.read(abyte0);
            class1 = defineClass(s, abyte0, 0, abyte0.length);
        }
        catch(Exception exception)
        {
            throw new ClassNotFoundException(s);
        }
        if(null != fileinputstream)
            try
```

```
            {
                fileinputstream.close();
            }
            catch(Exception exception1) { }
        return class1;
        Exception exception2;
        exception2;
        if(null != fileinputstream)
            try
            {
                fileinputstream.close();
            }
            catch(Exception exception3) { }
        throw exception2;
    }
```

c) Byte Code Assessment Conjectures and Discussion

protected findClass (string) returns: java.lang.Class

Assumption: The name of this function implies that it might be finding the class or its path. It returns java.lang.Class type, which may represent array or any primitive Java types (boolean, byte, short, int, etc.). The string parameter passed to it may be t he name of the class.

Method Bytecode:

- The method 'findClass' make following method class:
  - java.lang.String.replace (char, char) returns: string
  - java.lang.StringBuilder.append (string) returns: java.lang.StringBuilder
  - java.io.FileInputStream.available () returns: int
  - java.io.FileInputStream.read (byte []) returns: int
  - opencalculator.api.funcRunner.defineClass (string,byte [],int,int) returns: java.lang.Class
  - java.io.FileInputStream.close () returns: close

- **Other Information**:
  - A constant of type 'string' is declared and it is assigned a value ".class". The 'StringBuilder.append()' function is called before and after this string declaration.
  - A 'arraylength' instruction is found before the function 'opencalculator.api.funcRunner.defineClass()' is called.
  - Two 'if equals' conditions are found for the string constants which are loaded on the stack, and there are two 'FileInputStream.close()' function calls are found for each 'if equals' statements.
  - A 'java.lang.ClassNotFound' exception is thrown.

Discussion:

- The 'String.replace()' returns new string resulting from replacing all the occurrences of an old character with the new character. The 'StringBuilder.append()' function is called after the declaration of a string with the value '.class', which implies that ".class" is appended after a string which can be the name of a class. But it is difficult to guess why the method 'String.replace()' is called before appending ".class" to a string.
- The method 'FileInputStream.available()' returns the number of bytes that can be read from this file stream. The 'anewarray' instruction is found after this method call and this instruction is used to create an array of numeric type. It seems that an array is been created of the size returned by 'FileInputStream.available()' method.
- The method call 'FileInputStream.read (byte [] b)' reads up to b.length byte of data from this input stream into the array which has been created before.
- There is no method with the name 'defineClass()' in the 'funcRunner' class, so it is difficult to assume what it is used for.
- The 'if equals' conditions are comparing two strings, and a reference to the class is returned for a successful comparison. It seems that if the class is successfully found then its contents are retuned as a reference to it. It is difficult to judge that why there are two declarations of the 'if equals' conditions.
- The 'java.lang.ClassNotFound' exception might be thrown when the class is not found in the specified class path.

Conjecture: Once can make a conjecture about this method's overall functionality that it accepts the name of the class as a string parameter, finds it, and reads it byte by byte. But, it is difficult to judge the purpose of method calls 'String.replace()', 'opencalculator.api.funcRunner.defineClass()', and two 'if equals' conditions.

Confidence Rank: Medium
Value Ranking: Low

III. Method Name: FUNCtimeMs()

Class Name: **timeMs**

Confidence Rank: **High**

a) Byte Code

```
 0 new #2 <java/lang/Double>
 3 dup
 4 invokestatic #3 <java/lang/System.currentTimeMillis>
 7 l2d
 8 invokespecial #4 <java/lang/Double.<init>>
11 areturn
```

b) Decompiled Code

```
public timeMs()
{
}

public static Object FUNCtimeMs(Object aobj[])
{
    return new Double(System.currentTimeMillis());
}
```

c) Byte Code Assessment Conjectures and Discussion

public static FUNCtimeMs (java.lang.Object[]) returns: java.lang.Object
Assumption: The function's name suggests that it might be calculating the time / time interval in milliseconds. But its signature does not provide any extra information to support this assumption.

Method Bytecode:

- The 'FUNCtimeMs()' is making following method calls:
  - java.lang.System.currentTimeMillis() returns: long

- **Other Information**:
  - Total code length of this method is 12.

Discussion and Conjecture: The code length of this method is significantly less, which makes it easy to assess it. The only method call found is 'System.currentTimeMillis()', which returns the current time in milliseconds. This is the sufficient evidence to conclude that the function 'FUNCtimeMs()' returns current time in milliseconds.

Final Conclusion: The purpose of this class is to provide the current time in milliseconds.

Confidence Rank: High
Value Ranking: Low

IV. Method Name: countUtanParantes()

Class Name: OpenCalculator

Confidence Rank: Low

a) Byte Code

```
  0 aload_0
  1 aload_1
  2 invokespecial #22
<opencalculator/api/OpenCalculate.UtforFunktioner>
  5 astore_1
  6 aload_0
  7 aload_1
  8 invokespecial #23 <opencalculator/api/OpenCalculate.HittaMinustal>
 11 astore_1
 12 aload_0
 13 aload_1
 14 sipush 206
 17 sipush 206
 20 invokespecial #24 <opencalculator/api/OpenCalculate.UtforOperator>
 23 astore_1
 24 aload_0
 25 aload_1
 26 sipush 201
 29 sipush 203
 32 invokespecial #24 <opencalculator/api/OpenCalculate.UtforOperator>
 35 astore_1
 36 aload_0
 37 aload_1
 38 sipush 204
 41 sipush 205
 44 invokespecial #24 <opencalculator/api/OpenCalculate.UtforOperator>
 47 astore_1
 48 aload_0
 49 aload_1
 50 sipush 207
 53 sipush 210
 56 invokespecial #24 <opencalculator/api/OpenCalculate.UtforOperator>
 59 astore_1
 60 aload_0
 61 aload_1
 62 sipush 211
 65 sipush 212
 68 invokespecial #24 <opencalculator/api/OpenCalculate.UtforOperator>
 71 astore_1
 72 aload_0
 73 aload_1
 74 sipush 213
 77 sipush 213
 80 invokespecial #24 <opencalculator/api/OpenCalculate.UtforOperator>
 83 astore_1
 84 aload_0
 85 aload_1
 86 sipush 214
 89 sipush 214
 92 invokespecial #24 <opencalculator/api/OpenCalculate.UtforOperator>
 95 astore_1
 96 aload_1
 97 invokevirtual #7 <opencalculator/api/OpenCalculateSatts.size>
```

```
100 iconst_1
101 if_icmple 115 (+14)
104 new #15 <opencalculator/api/OCSyntaxError>
107 dup
108 ldc #25 <Opration saknas>
110 iconst_5
111 invokespecial #17 <opencalculator/api/OCSyntaxError.<init>>
114 athrow
115 new #26 <java/lang/Double>
118 dup
119 aload_1
120 iconst_0
121 invokevirtual #21 <opencalculator/api/OpenCalculateSatts.getValue>
124 invokespecial #27 <java/lang/Double.<init>>
127 areturn
```

b) Decompiled Code

```
private Double countUtanParantes(OpenCalculateSatts opencalculatesatts)
        throws Exception
    {
        opencalculatesatts = UtforFunktioner(opencalculatesatts);
        opencalculatesatts = HittaMinustal(opencalculatesatts);
        opencalculatesatts = UtforOperator(opencalculatesatts, 206,
206);
        opencalculatesatts = UtforOperator(opencalculatesatts, 201,
203);
        opencalculatesatts = UtforOperator(opencalculatesatts, 204,
205);
        opencalculatesatts = UtforOperator(opencalculatesatts, 207,
210);
        opencalculatesatts = UtforOperator(opencalculatesatts, 211,
212);
        opencalculatesatts = UtforOperator(opencalculatesatts, 213,
213);
        opencalculatesatts = UtforOperator(opencalculatesatts, 214,
214);
        if(opencalculatesatts.size() > 1)
            throw new OCSyntaxError("Opration saknas", 5);
        else
            return new Double(opencalculatesatts.getValue(0));
    }
```

c) Byte Code Assessment Conjectures and Discussion

private countUtanParantes (opencalculator.api.OpenCalculateSatts)
returns: java.lang.Double

Assumption: The English translation of the word ‘countUtanParantes’ is ‘count without/but parenthesis’. Its name implies that this method might be returning some value related to the parenthesis count.

Method Bytecode:

- It makes following method calls.

- opencalculator.api.OpenCalculate.UtforFunktioner(opencalculator.api.OpenCalculateSatts) returns: opencalculator.api.OpenCalculateSatts
- opencalculator.api.OpenCalculate.HittaMinustal(opencalculator.api.OpenCalculateSatts) returns: opencalculator.api.OpenCalculateSatts
- opencalculator.api.OpenCalculate.UtforOperator(opencalculator.api.OpenCalculateSatts,int,int) returns: opencalculator.api.OpenCalculateSatts
- opencalculator.api.OpenCalculateSatts.size() returns: int
- opencalculator.api.OpenCalculateSatts.getValue (int) returns: double

- **Other information:**
  - One conditional branch of 'if less the equals' is found with the object of the 'OCSyntaxError' class and a message string 'Opration saknas' is found. The English translation of this message is 'Operation missing'
  - This method returns a double value which is calculated by the 'opencalculator.api.OpenCalculate.getValue()' method.

Conjecture and Discussion:

- The long length of this method's bytecode makes it difficult to assess its internal operations.
- The method calls made by this method implies that this method is operating on results returned by the 'opencalculator.api.OpenCalculate.HittaMinustal()' and 'opencalculator.api.OpenCalculate.UtforFunktioner ()' methods. It is interesting to note that the method 'opencalculator.api.OpenCalculate.UtforOperator()' is invoked seven times whereas the other two methods are called only once.
- The purpose of method calls to 'opencalculator.api.OpenCalculateSatts.size()' and 'opencalculator.api.OpenCalculateSatts.getValue()' is difficult to judge.
- The error message 'operation missing' refers to the failure condition of this method and it implies that this method might be searching for some operators and in case they are not found then this message is displayed along with the object instantiation of the 'OCSyntaxError' class.

Confidence Rank: Low
Value Ranking: Low

APPENDIX E

UML Class Diagram

The following figure shows the high level UML class diagram of the Java program which reuses the methods of the ‘Open Calculator’ application.

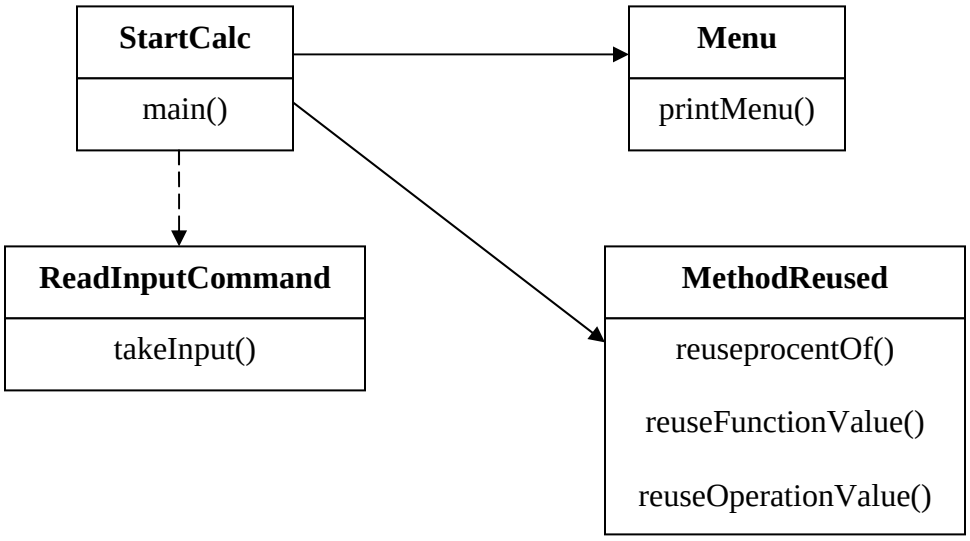

Figure 1: UML Class Diagram of the Java program that implements methods reuse.

The Class StartCalc is the starting point of the program. It creates an instance of the ‘Menu’ class and uses the method ‘takeInput()’ to accept the user input from the command line. The class ‘MethodResued’ has three member methods which are reusing the bytecode of the methods ‘procentOf()’, ‘getFunctionValue()’, and ‘getOperationValue()’. In order to make reuse of other methods of the ‘Open Calculator’ application, new methods can be added to the class ‘MethodResued’. Respective menu

options are required to be added in the ‘Menu’ class. The member methods of the class ‘MethodReused’ reuse the underlying functionalities of the above three methods, without writing them from scratch.

APPENDIX F

```
/*********************************************************************
 File Name:___StartCalc.java
 Class Name:___StartCalc
 Member Fields:__NONE
 Member Methods:_public static void main(String[])
   ********************************************************************/

   package opencalculator.api.func;
   import opencalculator.api.*;
   import java.io.*;

    public class StartCalc {

       public static void main (String[] args)_{

         Object temp[] = new Object[100];

         double percentage = 0.0;
         double trigResult = 0.0;
         double result = 0.0;
         double doubleValONE;
         double doubleValTWO;
         double doubleValTHREE;

         int intONE;
         int intTWO;
         int menuNumber = 0;

         do
         {
                    _
            Menu.printMenu();
            System.out.println("\n" + "Please one of the above option
                                    by entering the menu number:");

            try
            {
           menuNumber = integer.parseInt(ReadinputCommand.takeInput());

               MethodReused methodOperationValue =  new MethodReused();
               MethodReused methodFunctionValue =  new MethodReused();

               switch(menuNumber)
               {
```

```
                        case 1:
                        {
                          MethodReused percent = new MethodReused();
                        System.out.println("\n" + "To find the 'A'% of
                                                    some number 'B':");
                           System.out.println("Please enter value of
                                                           'A':");
                           doubleValONE =
                     Double.parseDouble(ReadinputCommand.takeInput());
                           System.out.println("\n" + "Please enter
                                                       value'B':");_
                           doubleValTWO =
                     Double.parseDouble(ReadinputCommand.takeInput());

                           percentage =
                     percent.reuseProcentOf(doubleValONE, doubleValTWO);

                           System.out.println("\nThe logic of the method
          'procentOf()' is being reused by exploiting its bytecode\n");
                           System.out.println("The "+ doubleValONE + "% of
                     " + doubleValTWO + " is = " + percentage + "\n");
                           break;
                        }

                     case 2:
                        {
                           System.out.println("\n" + "To find the
                                 Logarithm of 'A' with 10 as the base:");
                           System.out.println("Please enter value of
                                                           'A':");
                           doubleValONE = 0.0;
                           doubleValONE =
                     Double.parseDouble(ReadinputCommand.takeInput());
                           trigResult =
                methodFunctionValue.reuseFunctionValue(307,doubleValONE);

                           System.out.println("\nThe logic of the method
    'getFunctionValue()' is being reused by exploiting its bytecode\n");
                           System.out.println("The LOGARITHM of " +
          doubleValONE + " with 10 as the base is = " + trigResult + "\n");
                           break;
                        }

                     case 3:
                        {
                           System.out.println("\n" + "To find the SINE of
                                                           'A':");
                           System.out.println("Please enter value of 'A'
                                                      in degrees:");
                           doubleValONE = 0.0;
                           doubleValONE =
                     Double.parseDouble(ReadinputCommand.takeInput());
                           trigResult =
          methodFunctionValue.reuseFunctionValue(301,doubleValONE);
```

```
                        System.out.println("\nThe logic of the method
      'getFunctionValue()' is being reused by exploiting its
                                                    bytecode\n");
                        System.out.println("The SINE of " +
                  doubleValONE + " degrees is = " + trigResult + "\n");
                        break;
                     }

                  case 4:
                     {
                        System.out.println("\n" + "To find the COSINE
                                                    of 'A':");
                        System.out.println("Please enter value of 'A'
                                                    in degrees:");
                        doubleValONE = 0.0;
                        doubleValONE =
                  Double.parseDouble(ReadinputCommand.takeInput());
                        trigResult =
            methodFunctionValue.reuseFunctionValue(302,doubleValONE);

                        System.out.println("\nThe logic of the method
'getFunctionValue()' is being reused by exploiting its bytecode\n");
                        System.out.println("The COSINE of " +
                  doubleValONE + " degrees is = " + trigResult + "\n");
                        break;
                     }

                  case 5:
                     {
                        System.out.println("\n" + "To find the TANGENT
                                                    of 'A':");
                        System.out.println("Please enter value of 'A'
                                                    in degrees:");
                        doubleValONE = 0.0;
                        doubleValONE =

      Double.parseDouble(ReadinputCommand.takeInput());
                        trigResult =
            methodFunctionValue.reuseFunctionValue(303,doubleValONE);

                        System.out.println("\nThe logic of the method
'getFunctionValue()' is being reused by exploiting its bytecode\n");
                        System.out.println("The TANGENT of " +
                  doubleValONE + " degrees is = " + trigResult + "\n");
                        break;
                     }

                  case 6:
                     {
                        System.out.println("\n" + "To find the ARC SINE
                                                    of 'A':");
                        System.out.println("Please enter value of 'A'
                                                    in degrees:");
                        doubleValONE = 0.0;
                        doubleValONE =
```

```
                    Double.parseDouble(ReadinputCommand.takeInput());
                          trigResult =
              methodFunctionValue.reuseFunctionValue(304,doubleValONE);

                          System.out.println("\nThe logic of the method
'getFunctionValue()' is being reused by exploiting its bytecode\n");
                          System.out.println("The ARC SINE of " +
              doubleValONE + " degrees is = " + trigResult + "\n");
                          break;
                       }

                    case 7:
                       {
                          System.out.println("\n" + "To find the ARC
                                                 COSINE of 'A':");
                          System.out.println("Please enter value of 'A'
                                                    in degrees:");
                          doubleValONE = 0.0;
                          doubleValONE =
                    Double.parseDouble(ReadinputCommand.takeInput());
                          trigResult =
              methodFunctionValue.reuseFunctionValue(305,doubleValONE);

                          System.out.println("\nThe logic of the method
'getFunctionValue()' is being reused by exploiting its bytecode\n");
                          System.out.println("The ARC COSINE of " +
                    doubleValONE + " degrees is = " + trigResult + "\n");
                          break;
                       }

                    case 8:
                       {
                          System.out.println("\n" + "To find the ARC
                                                 TANGENT of 'A':");
                          System.out.println("Please enter value of 'A'
                                                    in degrees:");
                          doubleValONE = 0.0;
                          doubleValONE =
                    Double.parseDouble(ReadinputCommand.takeInput());
                          trigResult =
              methodFunctionValue.reuseFunctionValue(306,doubleValONE);

                          System.out.println("\nThe logic of the method
'getFunctionValue()' is being reused by exploiting its bytecode\n");
                          System.out.println("The ARC TANGENT of " +
                    doubleValONE + " degrees is = " + trigResult + "\n");
                          break;
                       }

                    case 9:
                       {

                          System.out.println("Please enter value of
                                                       'BASE':");
                          doubleValONE =
```

```
                    Double.parseDouble(ReadinputCommand.takeInput());
                          System.out.println("\n" + "Please enter value
                                                    of 'POWER/EXPNENT':");_
                          doubleValTWO =
                    Double.parseDouble(ReadinputCommand.takeInput());

                          result =
          methodOperationValue.reuseOperationValue(206, doubleValONE,
                                                          doubleValTWO);

                          System.out.println("\nThe logic of the method
    'getOperationValue()' is being reused by exploiting its bytecode\n");
                          System.out.println(doubleValONE + " to the
                power of " + doubleValTWO + " is = " + result + "\n");
                          break;

                       }

                    case 10:
                       {
                          System.out.println("\nThank you for using this
                                                    application.\n\n");
                          System.exit(1);
                       }
                 }

              }
                   catch(NumberFormatException nfex) {

                    System.out.println("\"" + nfex.getMessage() + "\" is
                                              not numeric \n");
                    System.exit(1);
                 }
           }
            while(true);
        }
     }
```

```
    /*****************************************************************
**
 File Name:___Menu.java
 Class Name:___Menu
 Member Fields:__NONE
 Member Methods:_public static void printMenu()
   *****************************************************************/
   package opencalculator.api.func;
   import opencalculator.api.*;

     public class Menu {

        public static void printMenu() {


         System.out.println("******* Wel-Come********");

         System.out.println("\n\n" + "Please seelct one of the
                                      following Options:");

         System.out.println("\n" + "1. Find Percentage:");
         System.out.println("2. Calculate Logarithm:");
         System.out.println("3. Calculate Sine:");
         System.out.println("4. Calculate Cosaine:");
         System.out.println("5. Calculate Tangent:");
         System.out.println("6. Calculate Arc Sine:");
         System.out.println("7. Calculate Arc Cosaine:");
         System.out.println("8. Calculate Arc Tangent:");
         System.out.println("9. Calculate Power:");
         System.out.println("10.Exit:");
      }
    }
```

```
 /*********************************************************************
File Name:___ReadinputCommand.java
Class Name:___ReadinputCommand
Member Fields:__NONE
Member Methods:_public static String takeInput()
*********************************************************************/

package opencalculator.api.func;
import opencalculator.api.*;
import java.io.*;
import java.util.*;
import java.lang.*;

 public class ReadinputCommand {

    public static String takeInput() {

      BufferedReader keyboard;
      String input = new String();

      try {
         keyboard = new BufferedReader(new
               InputStreamReader(System.in));
         System.out.flush();
         input = keyboard.readLine();
      }
          catch(IOException ioex) {
            System.out.println("Input error");
            System.exit(1);
         }

        return input;

   }
}
```

```
    /*****************************************************************
**
   File Name:        MethodReused.java
   Class Name:       MethodReused
   Member Fields:    NONE
   Member Methods:   public double reuseProcentOf(double, double ])
                     public double reuseFunctionValue(int, double)
                  public double reuseOperationValue(int, double, double)
   *****************************************************************/

   package opencalculator.api.func;
   import opencalculator.api.*;

    public class MethodReused {

       public double reuseProcentOf(double total, double value) {

         Object temp[] = new Object[100];
         double percentage;

         procentOf p = new procentOf();
         percentage = ((Double)procentOf.FUNCprocentOf(total,
                                  value,temp)).doubleValue();

         return percentage;
      }

       public double reuseFunctionValue(int caseNo, double input) {

         Object temp[] = new Object[100];

         double trigResult = 0.0;
         OpenCalculate calc = new OpenCalculate(temp);
         try
         {
            trigResult = ((Double)calc.getFunctionValue(caseNo,
                                                 input)).doubleValue();

         }
             catch(Exception e)
            {
                  System.out.println("Exeception Thrown: " + e);
            }

         return trigResult;
      }

       public double reuseOperationValue(int caseNo, double inputOne,
                                  double inputTwo) {

         Object temp[] = new Object[100];

         double result = 0.0;
         OpenCalculate calc = new OpenCalculate(temp);
         try
```

```
        {
           result = ((Double)calc.getOperationValue(caseNo, inputOne,
                 inputTwo)).doubleValue();

        }
            catch(Exception e)
           {
                       System.out.println("Exeception Thrown: " + e);
           }

        return result;
     }

      public void reuseFindClass(){

        Object temp[] = new Object[100];
        Class className;

        funcRunner classFinder = new funcRunner(temp);

        try{

           className =
           classFinder.findClass(".opencalculate.api.func.timeMs");

           System.out.println(className);
        }
            catch(Exception e)
           {
              System.out.println("Exeception Thrown: " + e);
           }
        }
}
```

APPENDIX G

1. jGRASP and Interactive UML class diagram

jGRASP [jGRASP] is a full-featured development environment that provides a great deal of software visualizations for the software comprehensibility improvement. Currently jGRASP provides three types of software visualizations: the Control Structure Diagram (CSD), Complexity Profile Graph (CPG), and UML class diagram.

jGRASP provides a convenient user-interface for generating an interactive UML class diagrams from Java class files. User can retrieve the basic architectural and dependency information of any Java program using this jGRASP feature. This feature can be used for various purposes such as during development, maintenance, and reverse engineering. This dependency information is gathered from the class files. The Figure 1 shows a screenshot of jGRASP that displays the UML class diagram of a *PersonalLibraryProject* [jGRASP]

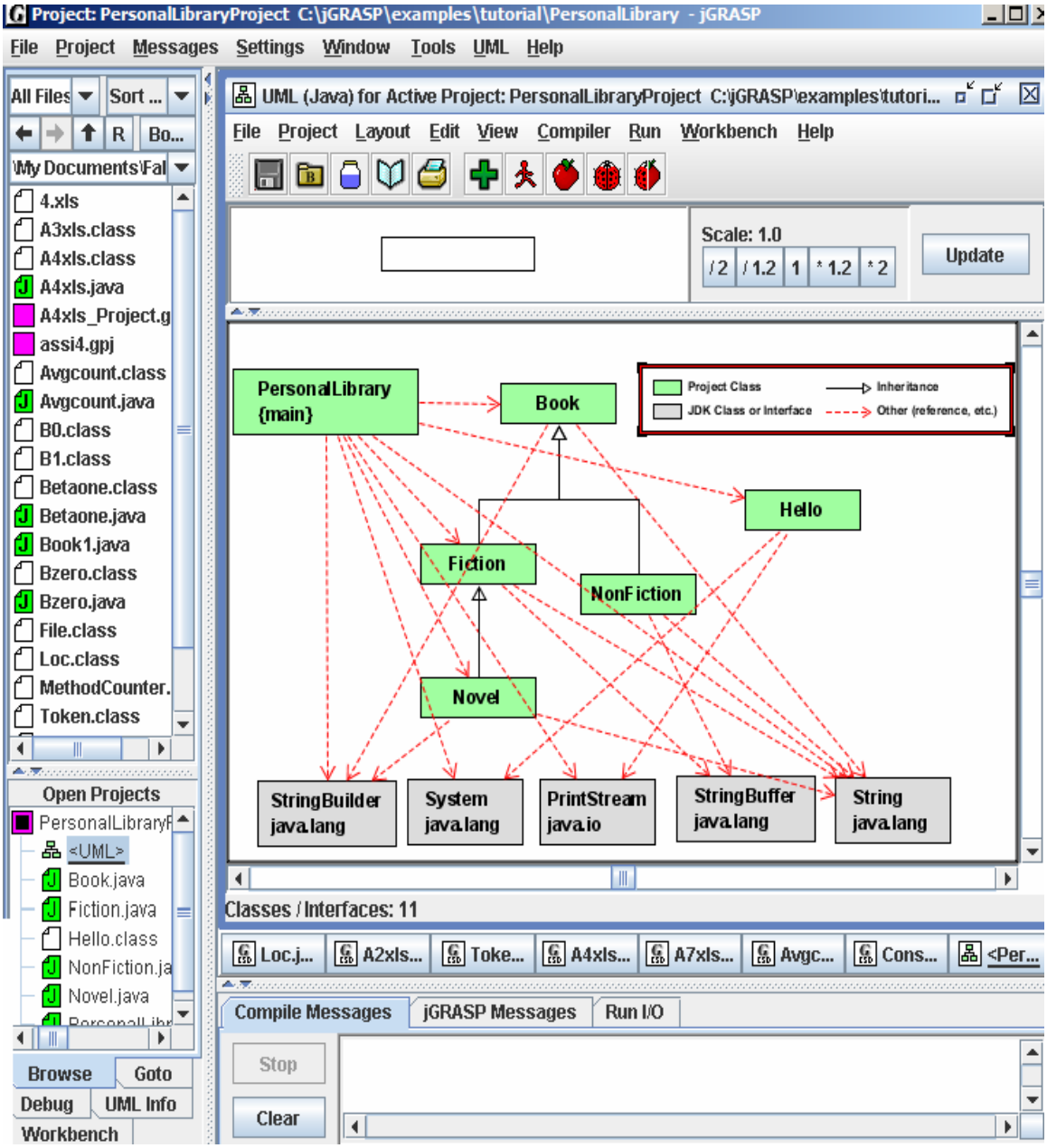

Fig. 1: jGRASP screenshots showing UML class diagram with dependency information [jGRASP]

2. jClassLib – Bytecode Viewer

jClassLib as a GUI utility that enables browsing the contents of the Java class file. The class details are displayed in accordance with the JVM specification. The figure 2

shows a jClassLib screenshot displaying the hierarchical view of the file structure in the left panel and the content of the selected element in the right pane.

The static details of each class element can be gathered suing this utility. The bytecode is displayed in assembly language, so that one can have better understanding of the code. It reveals information such as, what methods calls are made, value of the defined constants, details about the loops and the conditional instructions etc.

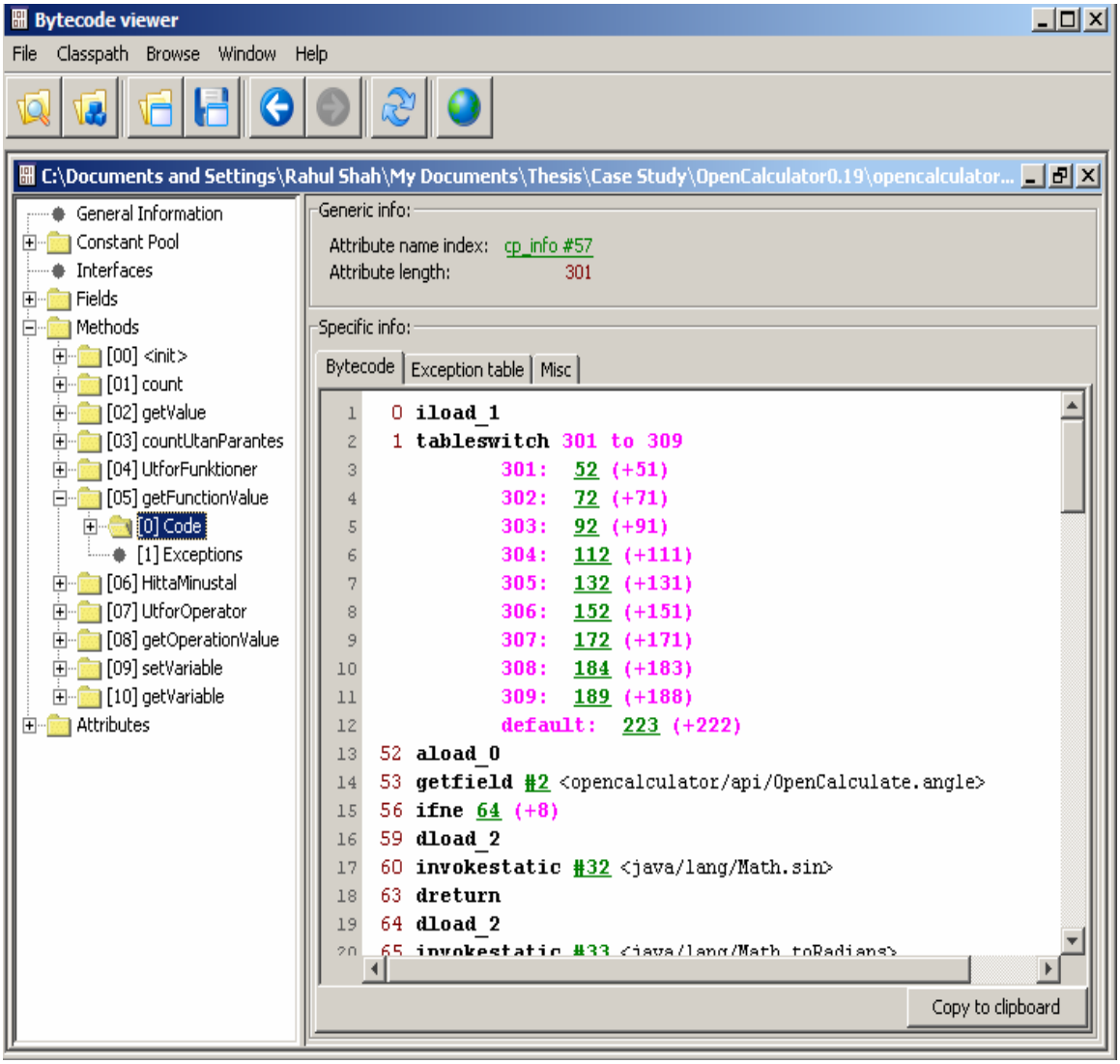

Fig. 2: jClassLib screenshot showing the details of 'OpenClaculate.class' file.

3. JAD – The Java Decompiler

Jad [Jad 1997] is a Java decompiler, which Java class files and converts them into Java source files which can be compiled again. It is a very fast and sophisticated reverse-engineering tool. It supports inner class definitions, anonymous implementations, and other Java language features. The Jad 1.5.8e version of the Jad has a command line interface.

4. FRHED – A Hex Editor

The ‘Free Hex Editor’ abbreviated as ‘FRHED’ is a binary file editor. It has a graphical user interface and following features:

- Cut, copy, and paste binary values.
- Allows entering or modifying the hex value in the main window.
- Bit manipulation.
- Automatically adjust bytes displayed per hexdump line to window width, or set bytes per line manually.

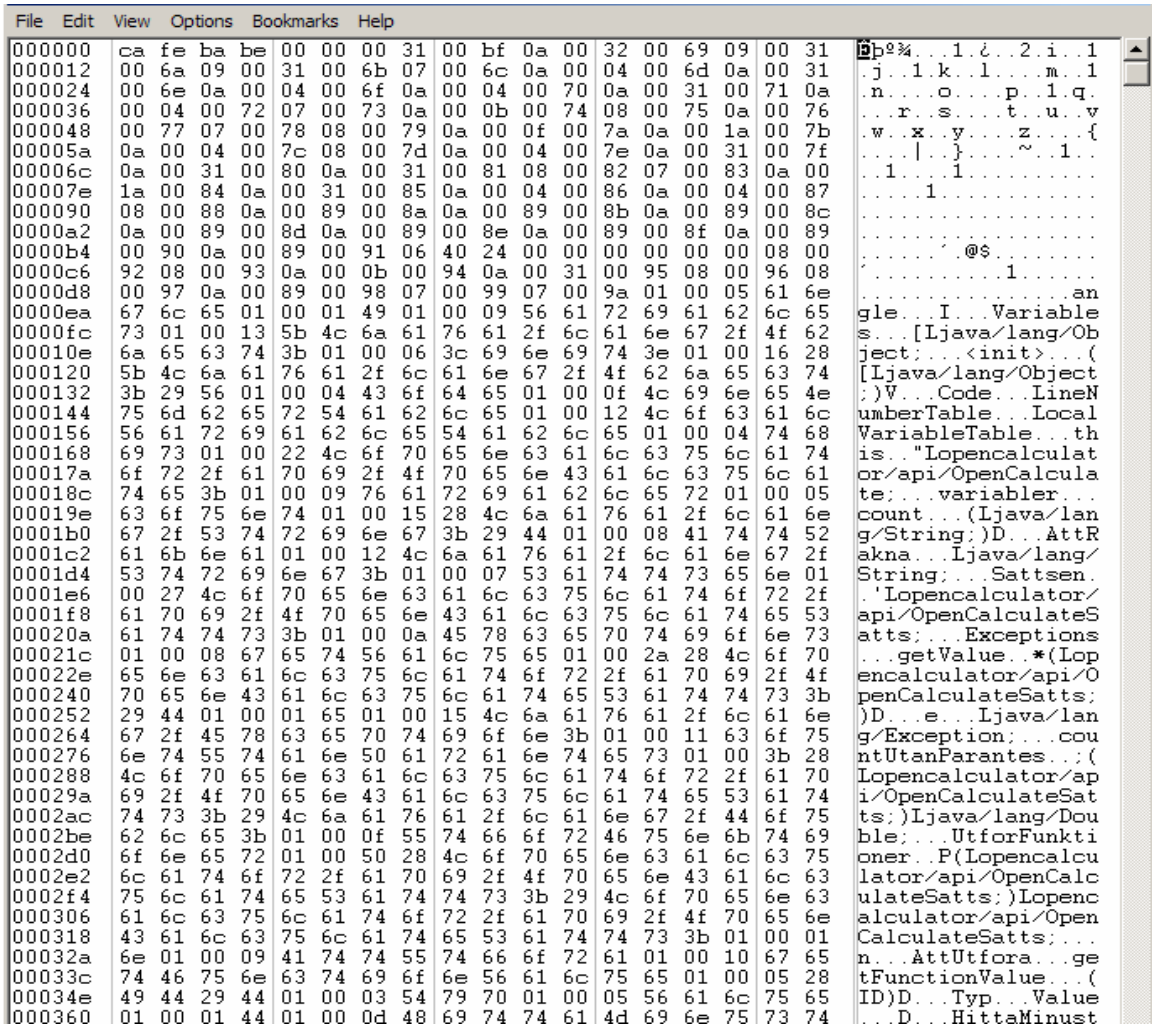

Fig. 3: FRHED – Hex Editor Screenshot

APPENDIX H

Class file structure

The class file structure is described in the Figure 1. Here u2 means a single byte, u2 means two bytes (or an *int*), and u4 means four bytes (or a *long*).

ClassFile {
u4 *magic*;
u2 *minor_version*;
u3 *major_version*;
u2 *constant_pool_count*;
cp_info constant_pool [*constant_pool _count* – 1];
u2 *access_flags*;
u2 *this_class*;
u2 *super_class*;
u2 *interfaces_count*;
u2 *interfaces* [*interfaces_count*];
u2 *fields_count*;
field_info fields [*fields_count];*
u2 *methods_count*;
method_info methods [*methods_count*];
u2 *attributes_count*;
attribute_info attributes [*attributes_count*];
}

Figure 1: Java Class File Format

Overview on each field of the Java class file

This literature review is an overview of the Class file format and the complete description of the Class file format can be found in JVM specifications [Lindholm and

Yellin 1999]. The following discussion gives a brief idea about each field found in *Java Class* file.

Magic number – 4 bytes

This identifies the class file format and verifies that JVM has loaded a Java class file. These four bytes will always be 0xCAFEBABE.

Minor version and Major version numbers – 2 bytes each

These two bytes identify *the minor and major version* numbers of the Class file format.

Constant pool count – 2 bytes

The value of *constant_pool_count* gives the total number of entries in the *constant pool table* plus one. The first item of the *constant pool* is reserved for internal JVM use, so the total value of *constant pool count* is one more.

Constant pool [constant_pool_count - 1] – variable

It is an array of variable length structure consists of several entries of string constants, class and field names, and other constants. It has format specified in Figure 2.

```
cp_info
{
        u1 tag;
        u1 info [ ];
}
```

Figure 2: Constant Pool Structure

Each item in the *constant_pool* table begins with 1-bye tag identifying the constant type and depending upon the value of the *tag*, size of *info* [ ] array is determined.

Table 1 summarizes all tags used in .class files.

Constant Type	Tag Value
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_Utf8	1

Table 1: Java Class File Tags

A tag value is 7 means that the next two bytes give an index into the *constant_pool* which is the name of the class; a value of 10 means two integers will come next; and a value of 1 indicates that the data to follow is string and so on. Thus, depending upon the value of the *tag*, the *info* [ ] has different structures. Some of the data structures discussed in the JVM specifications are described below.

a) Thus, when the tag value is 7, then following structure will be considered.

```
CONSTANT_Class_info {
                        u1 tag;
                        u2 name_index;
                    }
```

Figure 3: Structure for 'CONSTANT_Class_info'

The next two bytes are *name_index* that points to a valid index into the *constant_pool* array. The *constant_pool* entry at that index should be a CONSTANT_Utf8_info structure having name of the class or interface.

b) Fields, methods, and interface methods have the similar structural representation in the *constant_pool.*

```
CONSTANT_Methodref_info {
                    u1 tag;
                    u2 class_index;
                    u2 name_and_type_index;
                   }
```

Figure 4: Structur for ' CONSTANT_Methodref_info'

In case of the CONSTANT_Methodref_info, the tag value would be 10. The next two bytes represents the *class_index* having valid index number in the *constant_pool* table that should have an entry for CONSTANT_Class_info structure. This structure represents information of the class or interface type that contains the declaration of the method. The next two bytes are *name_and_type_index* that points to a structure of CONSTANT_NameAndType_info which has the name and description of the method.

c) CONSTANT_Utf8_info comprises about 59% of total structure in the *constant_pool* table [Antonioli and Pilz, 1998] and it is used to represent constant string values. The value of *tag* in this structure will always be 1. The next two bytes gives the length of the actual string that is followed next in the structure which is followed by the actual bytes.

```
CONSTANT_Utf8_info {
        u1 tag;
        u2 length;
        u1 bytes [length];
        }
```

Figure 5: Structure for 'CONSTANT_Utf8_info'

d) CONSTANT_NameAndType_info structure starts with the tag value 12. Next two bytes gives the index number of the *constant_pool* that ahs the CONSTANT_Utf8_info structure with the actual name of the method or the field. The last two bytes points to an array in the *constant_pool* having another CONSTANT_Utf8_info structure having description (signature) of the method or the valid field descriptor. Thus, this structure is used to store information about methods and fields without indicating which class or interface type they belong to.

```
CONSTANT_NameAndType_info {
        u1 tag;
        u2 name_index;
        u2 descriptor_index;
        }
```

Figure 6: Structure for 'CONSTANT_NameAndType_info'

Besides the above structures, *constant_pool* might have other structures depending upon the tag values. The other *constant types* and corresponding *tag values* are given in the Table 2.1 and more information can be found in the JVM specifications.

Access flags – 2 bytes

The next two bytes followed by the *constant_pool* are the *access_flags*. These bytes represent a mask of flags denoting the access permissions (access modifiers) used in class

and interface declarations. These modifiers are ACC_PUBLIC, ACC_FINAL, ACC_SUPER, ACC_INTERFACE, and ACC_ABSTRACT.

This class – 2 bytes

The value of *this_class* is a valid index of *constant_pool* having CONSTANT_Class_info structure describing *this* class.

Super class – 2 bytes

The value of *super_class* bytes can be zero or a valid index in *constant_pool* table having a CONSTANT_Class_info structure that describes the super class of this class. If the bytes representing *super_class* are zero then it means that the super class of the current class is *java.lang.Object*.

Interface count – 2 bytes

These two bytes represents the value of total number of superinterafces of this class or interface.

Interface [interface_count] – variable

This array contains one valid index into *constant_pool* for each interface implemented by the class. Each entry in the *constant_pool* contains CONSTANT_Class_info structure pointing to the name of the interface.

Fields count – 2 bytes

The *field_count* give the total number of fields declared in the class or the interface.

Field info [field_count] – variable

Following the *field_count* is an array of variable length structures one for each field declaration. Each structure reveals the field's information such as its name, type, access permissions, attributes' information etc. The structure might points to an array into constant_pool table.

```
field_info {
        u2 access_flags;
        u2 name_index;
        u2 descriptor_index;
        u2 attributes_count;
        attribute_info attributes [attributes_count];
        }
```

Figure 7: Structure for 'field_info'

Methods count – 2 bytes

The *methods_count* gives number of total methods or functions declared. This number includes the constructor method count.

Method info [methods_count] – variable

method_info is an array of variable length structures having complete description of each method decaled in this class or interface type. Following is the general format of a *method_info* structure.

```
method_info {
                u2 access_flags;
                u2 name_index;
                u2 descriptor_index;
                u2 attributes_count;
                attribute_info [attributes_count];
                }
```

Figure 8: Structure for 'method_info'

Thus, the *method_info* array has several pieces of information about each method, including method's name, descriptor, access permissions, a table of exceptions caught, the bytecode sequence etc.

Attributes count – 2 bytes

These two bytes gives the total number of attributes declared in the attribute array.

Attribute info [attributes_count] –variable

This is an array of variable length structures declaring attributes of this class file. The *attribute_info* structure contains information about its attribute's name, length, followed by the attribute themselves.

```
attribute_info {
        u2 attribute_name_index;
        u2 attribute_length;
        u2 info [attribute_length];
        }
```

Figure 9: Structure for 'attribute_info'

Some of the standard attributes are *SourceFile*, *Code*, *LineNumberTable* etc. Each standard attribute has predefined structure which is discussed in detail in the JVM specifications. [Lindholm and Yellin 1999]