Secure Coding Audit of Android Source Code

DISSERTATION

Submitted by

RAHUL.B CB.EN.P2CYS14010

in partial fulfillment for the award of the degree of

MASTER OF TECHNOLOGY
IN
CYBER SECURITY

TIFAC-CORE IN CYBER SECURITY
AMRITA SCHOOL OF ENGINEERING
AMRITA VISHWA VIDYAPEETHAM
COIMBATORE - 641 112

AUGUST 2016

Secure Coding Audit of Android Source Code

DISSERTATION

Submitted by

RAHUL.B   CB.EN.P2CYS14010

in partial fulfillment for the award of the degree of

MASTER OF TECHNOLOGY
IN
CYBER SECURITY

Under the guidance of

Prof. Prabhaker Mateti
Associate Professor
Computer Science and Engineering
Wright State University
USA

TIFAC-CORE IN CYBER SECURITY

AMRITA SCHOOL OF ENGINEERING

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE - 641 112

AUGUST 2016

AMRITA VISHWA VIDYAPEETHAM

AMRITA SCHOOL OF ENGINEERING,COIMBATORE -641 112

BONAFIDE CERTIFICATE

This is to certify that this dissertation entitled **“Secure Coding Audit of Android Source Code”** submitted by **RAHUL B. (Reg.No : CB.EN.P2.CYS14010)** in partial fulfillment of the requirements for the award of the **Degree of Master of Technology** in **CYBER SECURITY** is a bonafide record of the work carried out under my guidance and supervision at Amrita School of Engineering.

Dr. Prabhaker Mateti	**Dr. M. Sethumadhavan**
(Supervisor)	(Professor and Head)

This dissertation was evaluated by us on...............

INTERNAL EXAMINER EXTERNAL EXAMINERS

AMRITA VISHWA VIDYAPEETHAM
AMRITA SCHOOL OF ENGINEERING,COIMBATORE
TIFAC-CORE IN CYBER SECURITY

DECLARATION

I,RAHUL.B. (Reg.No: CB.EN.P2.CYS14010) hereby declare that this dissertation entitled **"Secure Coding Audit of Android Source Code"** is a record of the original work done by me under the guidance of **Prof. Prabhaker Mateti**, Associate Professor, Wright State University, and this work has not formed the basis for the award of any degree / diploma / associateship / fellowship or a similar award, to any candidate in any University, to the best of my knowledge.

Place : Coimbatore

Date : Signature of the Student

COUNTERSIGNED

Dr. M. Sethumadhavan
Professor and Head, TIFAC-CORE in Cyber Security

ABSTRACT

Android is the most widely used platform for smart phones. It gives support for a variety of file systems, process management, networking, virtual memory, root access (if rooted) and so on. In short, Android has the capability of a full operating system. This means that the attack surface is wide. The Android source code is freely available in the Internet named Android Open Source Project (AOSP). The AOSP is large in size and consists of code written in C, C++, and Java. The bad coding practice will open a door for an attacker to penetrate into the system. The primary goal of software security like confidentiality, integrity, and availability achieved through the implementation of security guidelines. Organization like CERT, OWASP and Oracle are providing secure coding guidelines. This thesis statically analyzes the Java source code files for compliance with CERT guidelines for secure coding. The full source code of the tool developed for detecting the code standard violation is available at the following link `https://github.com/rahulbpkl/Android-Java-Audit-Draft-1`.

Keyboards: Android, AOSP, Abstract Syntax Trees (AST), CERT, Secure Coding.

Contents

List of Tables

List of Figures

Acknowledgement

At the very outset, I would like to give the first honors to the **Almighty** who gave me the wisdom and knowledge to complete this dissertation.

I express my gratitude to my guide, **Prof. Prabhaker Mateti**, Associate Professor, Wright State University, USA, for his valuable suggestion and timely feedback during the course of this dissertation. It was indeed a great support from him that helped me successfully fulfill this work.

I would like to thank **Dr. M. Sethumadhavan**, Professor and Head of Department, TIFAC-CORE in Cyber Security, and **Mr. Praveen. K**, Assistant Professor, TIFAC-CORE in Cyber Security, for their constant encouragement and guidance throughout the progress of this dissertation.

I express my special thanks to my colleague, Anusuya, Karthik and Sridhar, who were with me from the starting of the dissertation, for the interesting discussions and suggestions. I convey special thanks to my friends for listening to my ideas and contributing their thoughts concerning the project. All those simple doubts from their part has also made me think deeper and understand about this work.

In particular, I would like to thank all the other faculties of TIFAC-CORE in Cyber Security and all those people who have helped me in many ways for the successful completion of this dissertation. And of course, thanks to my parents for helping me get where I am today.

1

Introduction

Android operating system is an open-source operating system developed by Google. It is the most widely used mobile platform today even though it came into the market in late 2008. Android was designed for users as well as developers. Users are provided visibility into how applications work, how to debug and control over those applications. Openness is the property that made people use Android in a wider range. This openness property has created users study, analyze and experiment their speculations with a minimal set-up. This has led to users exploiting holes in Android.

The primary goal of software security is to maintain the confidentiality, integrity, and availability. This aim is fulfilled via the implementation of security guidelines. Developing a secure software requires a good understanding of security guidelines. A fundamental component of secure software development is well-documented coding standards that support programmers to follow a uniform set of rules and recommendations. It will tell what he can do and what he can't. Once established, these rules and recommendations can be used as a metric to assess source code[Black et al. 2006].

Android is running on top of Linux kernel written in C. The application framework is written in Java. Android is a full-fledged networking device. A clever attacker targets the coding vulnerabilities. So it should satisfy the secure coding guidelines. There are many coding standards available in the market. OWASP, CERT and MISRA-C are some examples and most of them are dealing the same category with a different approach. Android doesn't satisfy most of these standards. There are many coding standard violations present in Android[Schmerl et al. 2016].

1.1 AOSP Code Example

The CERT secure coding standard[Long et al. 2011] implemented a rule related to file-related errors called FIO02-J. Detect and handle file-related errors. It tells that Java programs must check the return values of methods that perform file I/O. For example, if the programmer tries to delete one file it must check whether it deleted or not. There are many such kinds of violations in AOSP. Consider an example[Java/FIO02-J]:

```
if (entityFile.exists()) {
   entityFile.delete();
   }
```

This piece of code is from the file LocalTransport.java, location `frameworks/base/core/java/com/android/internal/backup/` in AOSP. Here entityFile is an object of File class and code entityFile.delete() is trying delete without checking the operation is full or not. If the file contains any sensitive information an attacker can use this or it may consume extra memory space. One solution is check the return value of delete(). The non compliant code is as follows[Java/FIO02-J]:

```
if (entityFile.exists()) {
   if (!entityFile.delete()) {
      //Add some code here
      }
   }
```

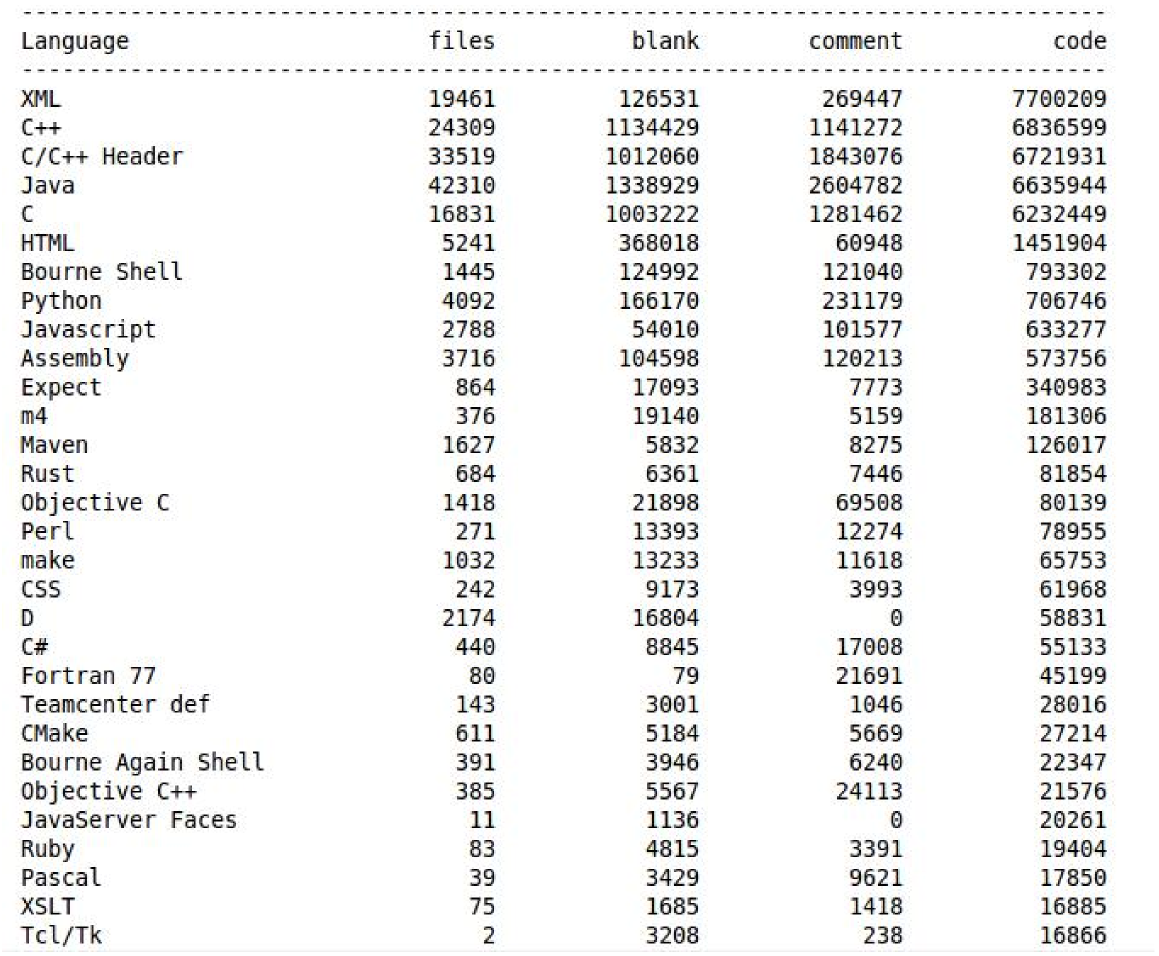
If entityFile.delete() fails it will enter to a block which will handle the failure. Another compliant solution uses the java.nio.file.Files.delete() method

```
try {
   Files.delete(file);
   } catch (IOException x) {
   System.out.println("Deletion failed");
   // Handle error
   }
```

Files.delete() will throw the NoSuchFileException, DirectoryNotEmptyException, IOException and SecurityException.

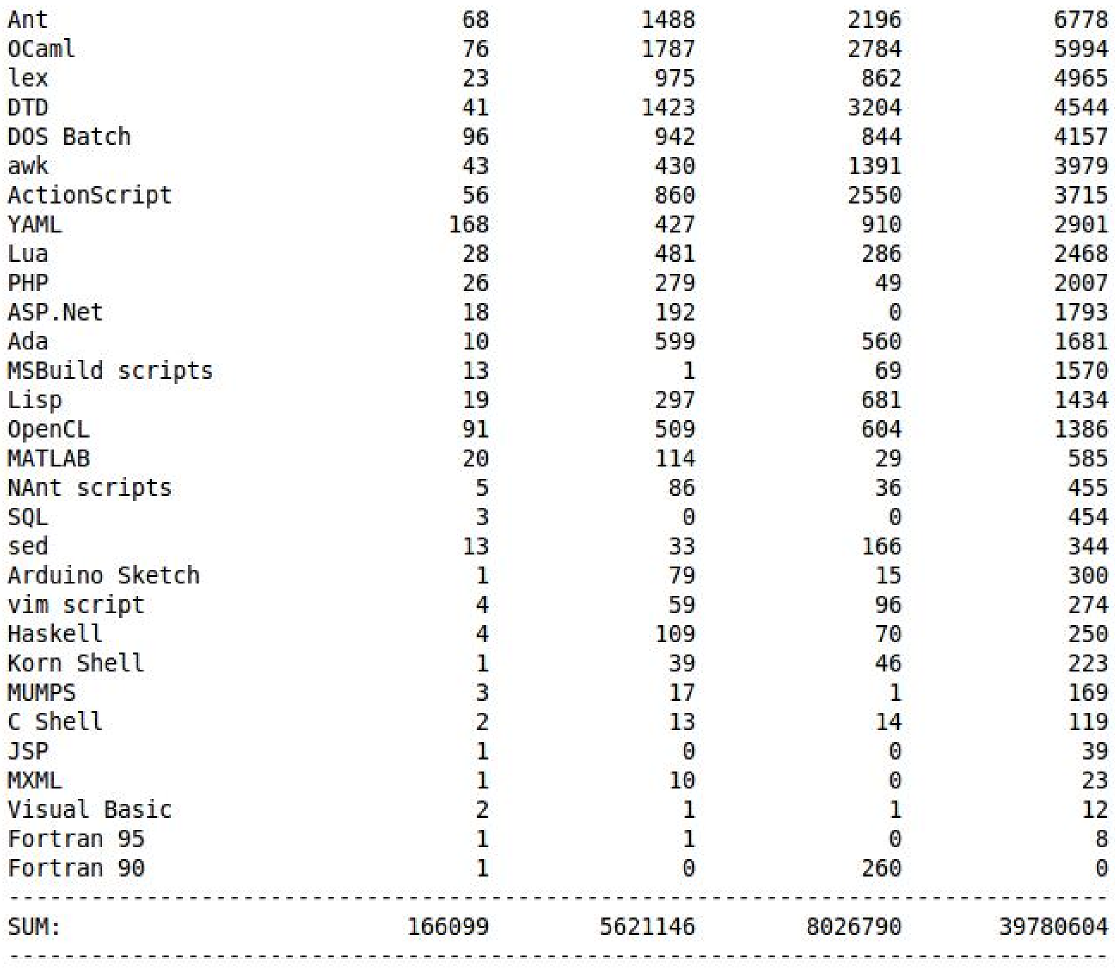
1.2 AOSP is Large

Android is a mobile OS but the project AOSP[Android 2015] is large in size. It includes all the SDK, NDK, Hardware related files, Framework, etc. The latest OS Marshmallow size is nearly 38GB in size and 35 million lines of codes. The AOSP total source lines of code (SLOC) and it's division is shown below.



```
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
XML                          19461         126531         269447        7700209
C++                          24309        1134429        1141272        6836599
C/C++ Header                 33519        1012060        1843076        6721931
Java                         42310        1338929        2604782        6635944
C                            16831        1003222        1281462        6232449
HTML                          5241         368018          60948        1451904
Bourne Shell                  1445         124992         121040         793302
Python                        4092         166170         231179         706746
Javascript                    2788          54010         101577         633277
Assembly                      3716         104598         120213         573756
Expect                         864          17093           7773         340983
m4                             376          19140           5159         181306
Maven                         1627           5832           8275         126017
Rust                           684           6361           7446          81854
Objective C                   1418          21898          69508          80139
Perl                           271          13393          12274          78955
make                          1032          13233          11618          65753
CSS                            242           9173           3993          61968
D                             2174          16804              0          58831
C#                             440           8845          17008          55133
Fortran 77                      80             79          21691          45199
Teamcenter def                 143           3001           1046          28016
CMake                          611           5184           5669          27214
Bourne Again Shell             391           3946           6240          22347
Objective C++                  385           5567          24113          21576
JavaServer Faces                11           1136              0          20261
Ruby                            83           4815           3391          19404
Pascal                          39           3429           9621          17850
XSLT                            75           1685           1418          16885
Tcl/Tk                           2           3208            238          16866
```

Figure 1.1: AOSP Lines of Code



```
Ant                              68          1488          2196          6778
OCaml                            76          1787          2784          5994
lex                              23           975           862          4965
DTD                              41          1423          3204          4544
DOS Batch                        96           942           844          4157
awk                              43           430          1391          3979
ActionScript                     56           860          2550          3715
YAML                            168           427           910          2901
Lua                              28           481           286          2468
PHP                              26           279            49          2007
ASP.Net                          18           192             0          1793
Ada                              10           599           560          1681
MSBuild scripts                  13             1            69          1570
Lisp                             19           297           681          1434
OpenCL                           91           509           604          1386
MATLAB                           20           114            29           585
NAnt scripts                      5            86            36           455
SQL                               3             0             0           454
sed                              13            33           166           344
Arduino Sketch                    1            79            15           300
vim script                        4            59            96           274
Haskell                           4           109            70           250
Korn Shell                        1            39            46           223
MUMPS                             3            17             1           169
C Shell                           2            13            14           119
JSP                               1             0             0            39
MXML                              1            10             0            23
Visual Basic                      2             1             1            12
Fortran 95                        1             1             0             8
Fortran 90                        1             0           260             0
-------------------------------------------------------------------------------
SUM:                         166099       5621146       8026790      39780604
-------------------------------------------------------------------------------
```

Figure 1.2: AOSP Lines of Code

1.3 Thesis Organization

The thesis discusses the design and implementation of AOSP source code auditing. The thesis is organized into seven chapters and Appendix A and Appendix B. Chapter 1 is Introduction. Chapter 2 provides background work undergone for the thesis. Chapter 3 contains the Problem Statement. Chapter 4 contains the Solution Architecture of the work. Chapter 5 Explains Results. Chapter 6 contains the Related Works did for the thesis. Chapter 7 contains the Evaluation of the implementations. Chapter 8 contains Conclusion and Future work of the thesis.

2

Background

This chapter describes the background of Secure Code Auditing for our work.

2.1 Parse Trees

Parsing is a process to determine how a string might be derived using productions of a given grammar[Aho and Ullman 1977]. It can be used to check whether or not a string belongs to a given language. The parser will produce parse tree often call it as derivation tree. In a derivation tree, all terminal, as well as non-terminals, can be present. If it is a full derivation tree, the leaf nodes are terminal symbols otherwise, some remain as non-terminals. The structure of the tree primarily shows how the derivation of the sentence was reconstructed for the given grammar. The structure can be treated with some "decorations" so that semantics can be deduced.

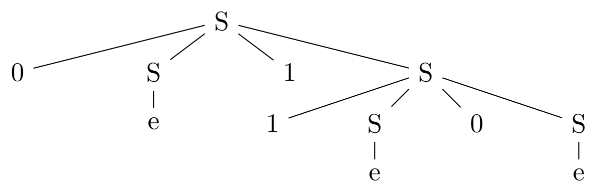


Figure 2.1: Full Derivation Tree of $S \rightarrow 011100$ (Grammar: $S ::= 0S1S \mid 1S0S \mid e$; e is empty)

2.1.1 Types of Parsing

Two basic approaches to parsing are top-down parsing and bottom-up parsing. In the top-down approach, a parser tries to derive the given string from the start symbol by rewriting nonterminals one by one using productions. The nonterminal on the left-hand side of a production is replaced by its right-hand side in the string being parsed. In the bottom-up approach, a parser tries to reduce the given string to the start symbol step by step using productions. The right-hand side of a production found in the string being parsed is replaced by its left-hand side.

2.1.2 Difficulties in Parsing

The main difficulty in parsing is nondeterminism. That is, at some point in the derivation of a string more than one productions are applicable, though not all of them lead to the desired string, and one can not tell which one to use until after the entire string is generated. For example in the parsing of "aababaa" discussed above, when S is at the top of the stack and a is read in the top-down parsing, there are two applicable productions, namely $S \to aSa$ and $S \to a$. However, it is not possible to decide which one to choose with the information of the input symbol being read and the top of the stack. Similarly, for the bottom-up parsing, it is impossible to tell when to apply the production $S \to a$ with the same information as for the top-down parsing. Some of these nondeterminism are due to the particular grammar being used and they can be removed by transforming grammars to other equivalent grammars while others are the nature of the language the string belongs to.

2.2 Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a tree representation of the source code written in a programming language. It is usually the result of the syntax analysis phase and serves as an intermediate representation of the program. AST starts with a root node and it contains child nodes. A terminal node in AST is either an identifier or a constant. The typical implementation of an AST makes large use of polymorphism. The nodes within the AST are usually implemented with a variety of classes, all deriving from a typical ASTNode class. For every syntactic construct within the language, there'll be a class for representing that construct within the AST, like ConstantNode, VariableNode, AssignmentNode, ExpressionNode, etc. A ConstantNode doesn't contain any children, an AssignmentNode must have two and an ExpressionBlockNode will have several children.

1. **Root Node** The node at the top of the tree is called root. If node n1 is root, it doesn't have any parent. There is only one root per tree and one path from root node to any node. The path is the connection between two nodes. In AST root node may be Classes, Enumdatatype, Interface, etc.

2. **Internal Node** If there are a parent and child for a node then it is called internal nodes. Node n1 is the parent of node n2 if n1 is an ancestor of node n2 and is connected to node n2. Node n2 is called a child of node n1. Nodes having the same parent are called siblings. In AST, operators/keywords are internal nodes.

3. **Leaf Node** Nodes with no children are called leaf nodes. In AST operands are located in the leaf node. In order to implement a single assignment statement(eg:x=10) there are three types of AST nodes, namely operator node(=), identifier node(x), literal node(10). The AST representation of an if-statement is shown in Figure 2.2.

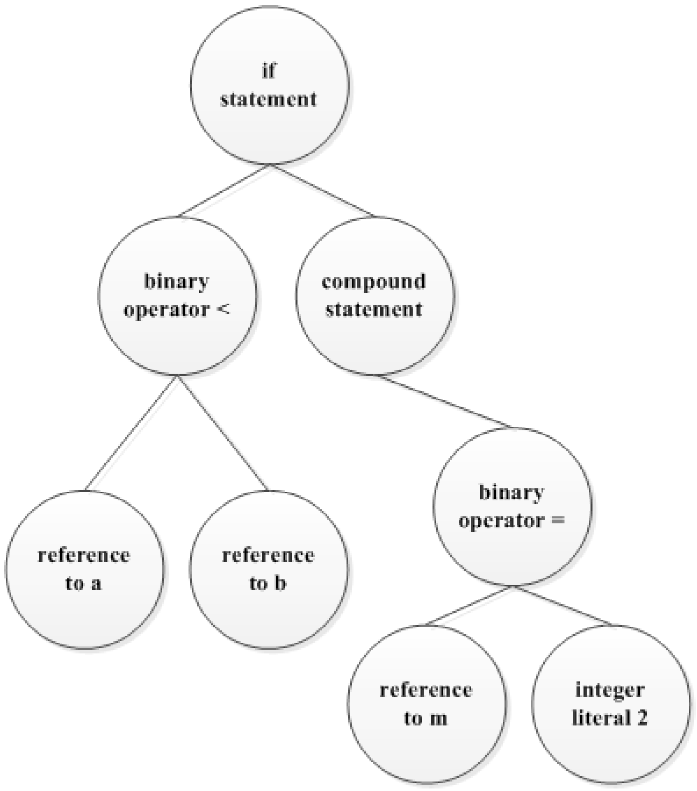
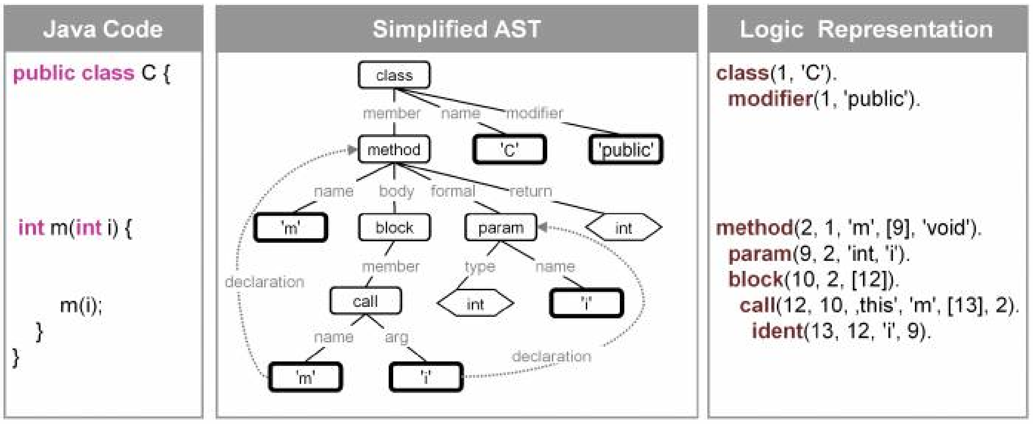


Figure 2.2: AST of if (a<b) m=2

There are many tools and APIs are there to generate AST. Eclipse JDT, PavaParser, JTransformer are some of the tools used to create AST from the given source code. Figure 2.3 shows the AST representation from JTransformer.



```
public class C {

 int m(int i) {

     m(i);
  }
}
```

```
class(1, 'C').
  modifier(1, 'public').

method(2, 1, 'm', [9], 'void').
  param(9, 2, 'int, 'i').
  block(10, 2, [12]).
    call(12, 10, ,this', 'm', [13], 2).
      ident(13, 12, 'i', 9).
```

Figure 2.3: Java src, its AST, and Stored Representation using JTransformer [iai.uni bonn.de 2011]

2.2.1 Visitor Pattern

A compiler parses a program and represents it as an AST. It has various kinds of nodes like Assignment, Variable Reference, Arithmetic Expression nodes, etc. Operations that one would really like to perform on the AST are type checking, code generation, variable declaration, syntax analysis, etc. These actions should handle each type of node separately. One solution is to define each operation in a separate node class(Figure 2.4). The problems with this approach are firstly it can be a confusing and time-consuming process. Secondly adding new operations require changes to all of the node classes. To solve this problem an alternative method visitor pattern is used[Wikipedia 2009].

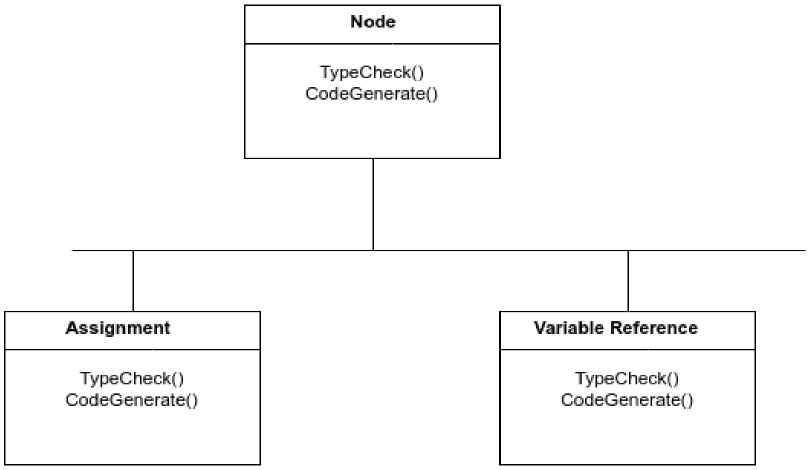


Figure 2.4: Operations in separate node class

The Visitor pattern[Wikipedia 2009] was introduced to address the above problem. It lets us to define a new operation without changing the classes of the elements on which it operates. Instead of spreading all the code for a given traversal throughout the nodes classes, the code is concentrated in a particular traversal class(visitor class). That is, desired operation is encapsulated in a separate object, called a visitor. Then the visitor object will traverse the nodes of the tree. The structure of Visitor Pattern is shown in Figure 2.5.

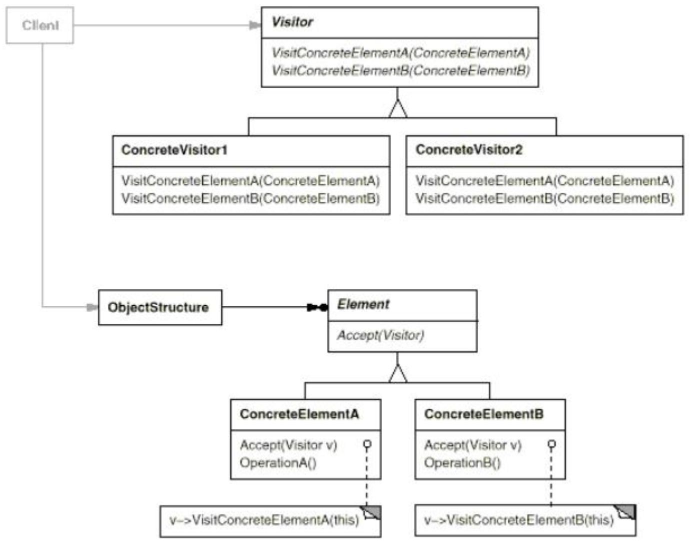

Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)

Figure 2.5: Structure of Visitor Pattern
[Wikipedia 2009]

The participants classes in this pattern are:

1. **Visitor** It is used to declare the visit operations for all the types of visitable classes. Normally, the name of the operation remains the same and these operations are differentiated by the method signature. The input object type decides which of the method to be called. Visitor is implemented with the help of interface or an abstract class.

2. **ConcreteVisitor** For each type of visitor, all the visit methods declared in abstract visitor must be implemented. Each Visitor is responsible for different operations. ConcreteVisitor implements each operation declared by Visitor. Each operation executes a piece of the algorithm defined for the corresponding class of the object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state.

3. **Visitable** This is the entry point which enables an object to be “visited” by the visitor object.

Each object from a collection should implement this abstraction in order to be able to visit.

4. **ConcreteVisitable** Those classes implements the Visitable interface or class and defines the accept operation. The visitor object is passed to this object using the accept operation.

5. **ObjectStructure** This is a class containing all the objects that can be visited. It offers a mechanism to iterate through all the elements. This structure is not necessarily a collection. It can be a complex structure, such as a composite object.

The participants classes are shown in Figure 2.6

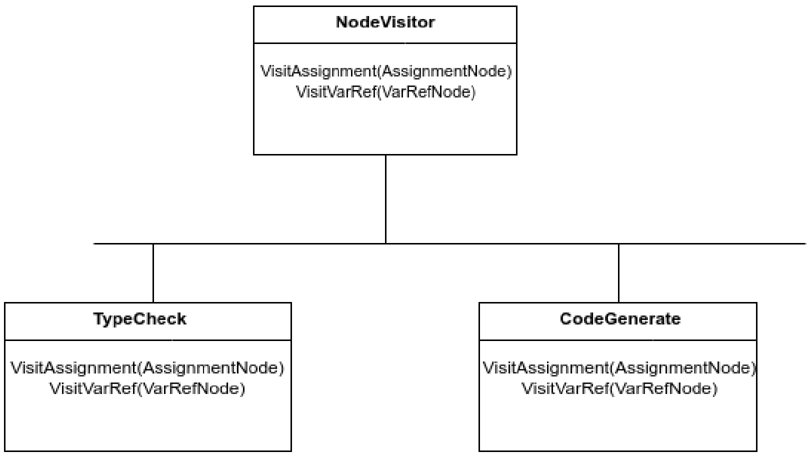


Figure 2.6: Visitor Pattern

2.2.2 Advantages

Some benefits of Visitor Pattern are:

1. It allows adding operations to a structure without changing the structure itself.

2. Adding new operations is relatively easy.

3. The code for operations performed by the Visitor is centralized so editing and adding new code is easy.

2.2.3 Applications of Visitor Pattern

1. It is applicable when similar operations have to be performed on objects of different types grouped in a structure.

2. There are many distinct and unrelated operations needed to be performed. Visitor pattern allows creating a separate visitor concrete class for each type of operation. It separates this operation implementation from the object structure.

3. The object structure is not likely to be changed but is very probable to have new operations which have to be added. Since the pattern separates the visitor (representing operations, algorithms, behaviors) from the object structure it's very easy to add new visitors as long as the structure remains unchanged.

2.3 Secure Coding Guidelines

MITRE has listed almost 700 different kinds of software weaknesses in their CWE project[MITRE 2012]. These are all different ways that software developers can make mistakes that lead to vulnerability. Most of the developers are not aware of these weaknesses. In olden days, software industry didn't care much about secure coding standards because the lack of time, lack of security standards and no extra payment from the client, but currently, programmers started developing software systems by following secure coding standards. Secure code review is the process of auditing the source code for an application to verify that the proper security controls are present and they are invoked in the right places. Security code review is a way of ensuring that the application has a self-defending capability.

Building secure software requires a basic understanding of security guidelines. The goal of software security is to maintain the confidentiality, integrity, and availability of information resources in order to enable successful operations. This goal is accomplished through the implementation of security controls. An essential element of secure software development is well-documented and enforceable coding standards. Coding standards encourage programmers to follow a uniform set of rules and guidelines determined by the requirements of the project and organization, rather than by the programmer's familiarity or preference. Once established, these standards can be used as a metric to evaluate source code. There are many secure coding standards present in the software industry. Some of them are,

2.3.1 MISRA

MISRA[Association et al. 2008] coding standards have been adopted by industries for developing safety-related electronic systems in road vehicles, telecom, aerospace and other embedded systems. They have separate guidelines for C and C++. MISRA is not an open standard, so the implementers have to pay for the guideline documents. MISRA-C:2012 contains 143 rules and 16 directives. Each of the rules is classified as “mandatory”, “required”, or “advisory” based on the severity. Another way the rules are classified as Decidable or Undecidable. Coverity Static Analysis, Eclair, GrammaTech, Goanna are some tools for verifying MISRA standard.

2.3.2 CERT Secure Coding

The CERT[Seacord 2008] Secure Coding Standard provides rules and recommendations (collectively called guide- lines) for secure coding in the programming language. It has separate guidelines for C, C++, Java and Android Java. The goal of these rules and recommendations is to develop safe, reliable, and secure systems. There are for C 194 Rules and 77 Recommendations and for Java 122 Rules and 180 Recommendations. Coverity Static Analysis, Eclair, Rosechecker are some tools for verifying CERT standard.

2.3.3 OWASP

The OWASP is an online community which creates a coding standard, documentation, tools, for web application security which deals specifically with the security of websites, web applications, and web services. OWASP supports various security-related projects, one of the most popular is the OWASP Top 10[OWASP 2010]. It is a famous web application security by identifying some of the most critical risks, provides examples, and offers suggestions on how to avoid it. The Top 10 vulnerabilities are Injection, Broken Authentication and Session Management, Cross Site Scripting, Insecure Direct Object References, Security Misconfiguration, Sensitive Data Exposure, Missing Function Level Access Control, Cross Site Request Forgery (CSRF), Using Components with Known Vulnerabilities, Unvalidated Redirects and Forwards. These Top 10 is referenced by many standards, books, tools, and organizations.

2.4 Static Analysis of Source Code

The analysis of code without executing is called static analysis[Chess 2007]. Automated tools simplify the overhead of a static code analysis. By examining the code itself, static analysis tools can often point to the root cause of a security problem. This is particularly important for making sure that vulnerabilities are fixed before the program is run for the first time. In secure coding static analysis tools are now common one which will catch the coding standard violations before execution. While typing itself some tools integrated with the text editor figure out the error. It will help to prevent the common coding vulnerabilities like buffer overflow, cross-site scripting, SQL injection etc.

There are some problems with static analysis tools, the main ones are false positive and false negatives. False positive is a problem reported in a program when no problem actually exists. A large number of false positives can cause real difficulties. It will increase the workload of the programmer, unwanted wastage of time and effort for correction. With a false negative, a problem exists in the program, but the tool does not report it. The penalty for a false negative is more danger. What is the severity of a particular violation, it will happen in future. So the false negatives are much worse. All static analysis tools are guaranteed to produce some false positives or some false negatives or both.

Static analysis tool address the following problems:

2.4.1 Type Checking

The most widely used form of static analysis is type checking. Many programmers dont give type checking much thought. Type checking eliminates entire categories of programming mistakes. For example, it prevents programmers from accidentally assigning integral values to object variables. By catching errors at compile time, type checking prevents runtime errors.

2.4.2 Style Checking

Style checkers are also static analysis tools. Pure style checkers enforce rules related to whitespace, naming, deprecated functions, commenting, program structure, and the like. Because many programmers are fiercely attached to their own version of the good style, most style checkers are quite flexible about the set of rules they enforce. The errors produced by style checkers often affect the readability and the maintainability of the code but do not indicate that a particular error will occur when the program runs.

2.4.3 Program Understanding

Program understanding tools help users make sense of a large codebase. Integrated development environments (IDEs) always include at least some program understanding functionality. More advanced analysis can support automatic program-refactoring features, such as renaming variables or splitting a single function into multiple functions. Higher-level program understanding tools try to help programmers gain insight into the way a program works.

2.4.4 Bug Finding

The purpose of a bug finding tool is to find the violation of rules, a bug finder simply points out places where the program will behave in a way that the programmer did not intend. Most bug finders are easy to use because they come pre-defined with a set of rules that describe patterns in code that often indicate bugs.

2.4.5 Security Review

Security focused static analysis tools use many of the same techniques found in other tools, but they are more focused on identifying security problems. There are many secure coding standards (CERT, OWASP etc)and they defined a set of rules. Security review will verify whether the programmer followed the coding standard.

3

Problem Statement

Our goal is to take the AOSP Marshmallow files and verify that they obey the security coding compliance rules, developed by CERT. We expect our solution to work for other codebases of Android: CyanogenMod 13, Paranoid Android, XenonHD, etc. However, ours being a research project, it is driven by a focus on technique. We intend to focus only on Java. This amounts to roughly the Android Framework, and several essential APKs such as Camera, Email, Gallery, etc.

3.1 Explanation of Violations Discovered

This is a goal. When a code fragment is flagged, it is crucial that it contains explanations that an average programmer can understand. The explanation contains what violation is there and where(line number), and a description for the user with a compliant and non-compliant code. If the code size is large like AOSP so the report may contain lots of explanation. A well-structured report will help the programmer to ease the task. The changes needed to bring the code into compliance will almost always have to be made by human(interactive) intervention. This will be a research issue.

3.2 Minimal Number of False Positives

This is a goal. A false positive is a problem reported in a program when no problem actually exists. A large number of false positives can cause real difficulties. The penalty for a false positive is the amount of time wasted while reviewing the result. We also need to make sure that anything flagged is almost always non-compliant. This will be a research issue.

3.3 Dealing With Code Changes

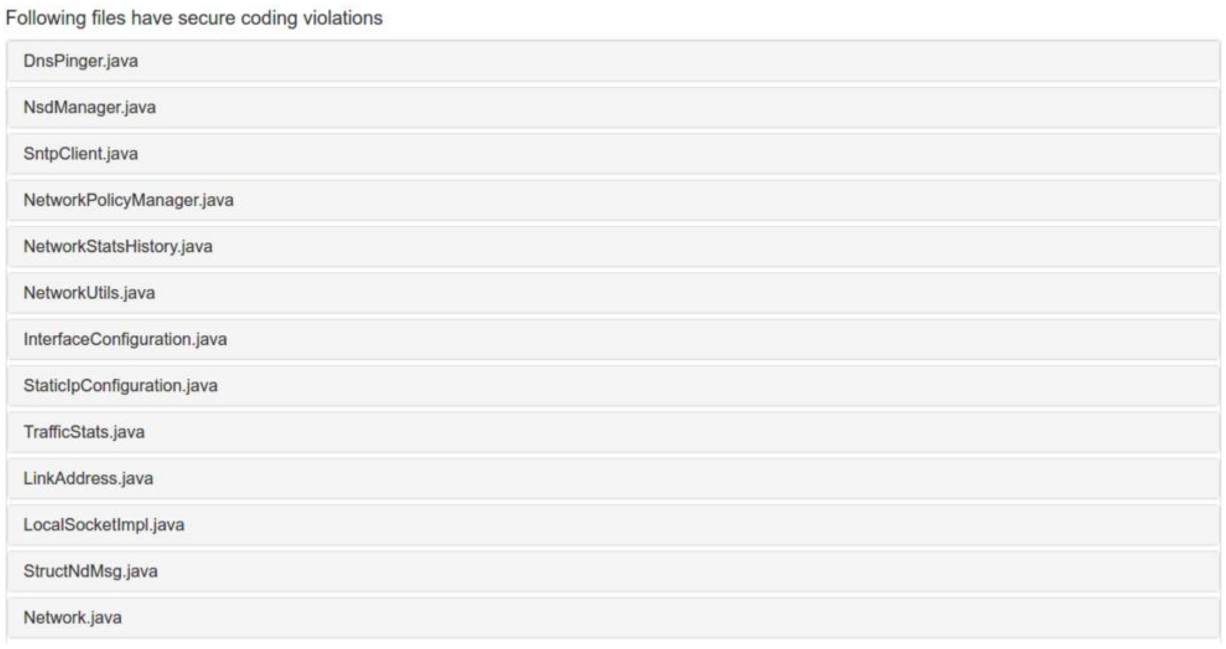
The AOSP changes approximately once or twice a year; so, we would like it to be checked automatically. Occasionally, the changes encompass significant changes in the base language. Until recently, the Java used in AOSP was Java-6. E.g., Android Nougat (yet to be fully released as of July 2016) is based on Java-8 and uses lambda expressions and other such features.

3.4 AOSP Over the Linux Kernel

AOSP includes the Linux kernel, written in C. CERT providing rules for Android C also. But we excluded it from the scope of this thesis.

3.5 A Report Sheet Per Component

We will produce a code compliance verification report sheet per component. A speculative example is shown figure 3.1. Each file name contains a link to the explanation of the violation and the contents will expand and collapse with the help of Bootstrap and Jquery.



Following files have secure coding violations

DnsPinger.java
NsdManager.java
SntpClient.java
NetworkPolicyManager.java
NetworkStatsHistory.java
NetworkUtils.java
InterfaceConfiguration.java
StaticIpConfiguration.java
TrafficStats.java
LinkAddress.java
LocalSocketImpl.java
StructNdMsg.java
Network.java

Figure 3.1: Report sheet per component

4 Solution Architecture

Our aim is to audit the ASOP with a maximum number of rules and in an efficient way. This chapter describes the architecture of our solution.

4.1 Java Source Code Compiled to AST

An AST is a tree representation of the source code and usually the result of the syntax analysis phase and serves as an intermediate representation of the program. The syntax analysis is the second phase of the compiler design and another important lexical analysis phase is there before it. In the lexical analysis, the lexical analyzer takes the source code from language preprocessors that are written in the form of text file. The lexical analyzer chunks these syntaxes into a series of tokens, by excluding any whitespace or comments in the source code and passes the data to the syntax analyzer when it demands. If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer can identify tokens with the help of regular expressions and pattern rules. It works closely with the syntax analyzer. In syntax analysis phase syntax analyzer or parser that groups sequences of tokens from the lexical analysis phase into phrases each with an associated phrase type and it produces the parse tree and syntax tree. A lexical analyzer uses regular expressions and pattern rules for identifying tokens, but a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. It cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata. To generate the AST the program or tool needs to do the first two complex phases of the compiler. Generating an AST from the scratch is a difficult and time-consuming process. There are many tools to do the equivalent of the following (see Section 6.2.1).

4.2 User Interface

Our user interface is command line based which will accept some arguments and files as input and produce the violations list and explanation of non-compliance as output. It will run on both Windows and Linux operating system. The arguments will help to use the tools as user's wish. The format is:

```
<rast> <argument> <filename/directory path> <optional arguments>
```

The arguments are:

```
-b For batch conversion
-s For single file conversion
-h For help page
-f For semi auto fix
<optional arguments> For Checking/Fixing a particular rule.
```

4.3 CERT Rules

CERT rules are written in English with a description, compliant and non-compliant code. CERT doesn't provide any algorithm to detect the violation. We implemented an algorithm to detect the code compliance. We focused on Android rules (see Section 4.6). There are about 82 Android Java rules. One example of CERT Android Java is LCK01-J. Do not synchronize on objects that may be reused. The rule says that abuse of synchronization primitives will make concurrency issues. Synchronizing on objects that may be reused can result in deadlock and nondeterministic behavior. Consequently, programs must never synchronize on objects that may be reused.

4.4 Rule Applicability Condition

Sometimes the CERT non-compliance may happen on one line, otherwise, a block of code depends on the rule. To detect the applicability visit the corresponding nodes and do a pattern match. Consider an example rule, ERR07-J says that don't throw RuntimeException, Exception, or Throwable. This violation can happen in a single line of the source code file. To detect this violation the method is to check any class object creation for RuntimeException, Exception, or Throwable is present in throw block.

4.5 Explanation of Non-Compliance

AOSP is non-compliant to CERT, consider the example in the section 4.4 (ERR07-J) is says that don't throw RuntimeException, Exception, or Throwable. The tool reported so many violations of ERR07-J in AOSP. Consider a violation from frameworks/base/core/java/android/net/NetworkUtils.java line number 347 it is trying to throw runtime exception. The code block is as follows:

```
public static InetAddress getNetworkPart(InetAddress address, int prefixLength) {
byte[] array = address.getAddress();
maskRawAddress(array, prefixLength);
InetAddress netPart = null;
try {
   netPart = InetAddress.getByAddress(array);
} catch (UnknownHostException e) {
  throw new RuntimeException("getNetworkPart error - " + e.toString());
}
   return netPart;
}
```

The code throw new RuntimeException("getNetworkPart error - " + e.toString()); is violating ERR07-J. To avoid this violation the programmer must check the null value and call NullPointerException. The code will be

```
if (netPart == null) {
   throw new NullPointerException();
}
else
{
   return netPart;
}
```

4.6 Android Secure Coding Standard of CERT

CERT have separate secure coding standard for Android[Schiela 2014]. Several of these rules have been updated in 2016. Android uses C, C++ and Java; however, we explore further only the Java specifics.

4.6.1 DCL00-J. Prevent Class Initialization Cycles

```
public class Cycle {
 static Cycle cy = new Cycle();
 static int deposit = 200;
 int balance;

 public Cycle() {
  balance = deposit - 10;
 }

 public static void main
  (String[] args) {
   System.out.println("balance = " + cy.balance);
  }
}
```

The above code is noncompliant. At the time `cy` is initialized (by the constructor), `deposit's` value is 0. So balance is set to -10. The cy initialization should be done after that of deposit[CERT/DCL00-J].

4.6.2 DCL01-J. Do Not Reuse Public Identifiers from the Java Standard Library

```
Class Vector {
 private int val = 1;

 public boolean isEmpty() {
   if (val == 1) { // Compares with 1 instead of 0
    return true;
   } else {
    return false;
   }
  }
  // Other functionality is same as java.util.Vector
}

// import java.util.Vector; omitted
public class VectorUser {
 public static void main(String[] args) {
  Vector v = new Vector();
  if (v.isEmpty()) {
   System.out.println("Vector is empty");
  }
 }
}
```

Suppose we wrote our own class and named it as `Vector`. This `Vector` shadows the class name from `java.util.Vector`[CERT/DCL01-J].

4.6.3 DCL02-J. Do Not Modify the Collection's Elements During an Enhanced for Statement

```
import java.util.*;
public class EnhancedFor {
 public static void main
  (String[] args) {
   List < Integer > lst =
    Arrays.asList(new Integer[] {
     11,
     12,
     13,
     14
    });

   for (Integer i: lst) {
    if (i < 13) {
     i = new Integer(99);
    }
    System.out.println("New: " + i);
   }

   for (Integer i: lst) {
    System.out.println("Old: " + i);
   }
  }
}
```

In the above, the first enhanced-for loop appears to change the first few items of the `lst` to 99, but as the second for-loop demonstrates, did not[CERT/DCL02-J].

4.6.4 EXP02-J. Do Not Use the Object.Equals() Method to Compare Two Arrays

```
public class ArrayClass {
 public static void main(String args[]) {
  int[] arr1 = new int[20];
  int[] arr2 = new int[20];
  if (arr1.equals(arr2)) {
   System.out.println("ARRAYS ARE SAME");
  }
 }
}
```

The above code uses the Object.equals() method to compare two arrays arr1 and arr2[CERT/EXP02-J].

4.6.5 ERR00-J. Do Not Suppress or Ignore Checked Exceptions

```
class Foo implements Runnable {
 public void run() {
  try {
   Thread.sleep(1000);
  } catch (InterruptedException e) {
   // Ignore
  }
 }
}
```

This code prevents callers of the run() method from determining that an interrupted exception occurred. Consequently, caller methods such as Thread.start() cannot act on the exception[CERT/ERR00-J].

4.6.6 ERR01-J. Do Not Allow Exceptions to Expose Sensitive Information

```
class ExceptionExample {
 public static void main(String[] args) throws FileNotFoundException {
  // Linux stores a user's home directory path in
  // the environment variable $HOME, Windows in %APPDATA%
  FileInputStream fis =
   new FileInputStream(System.getenv("APPDATA") + args[0]);
 }
}
```

An attacker tries to reconstruct the underlying file system by repeatedly passing false path names to the program because when a requested file is absent, the FileInputStream constructor throws a FileNotFoundException[CERT/ERR01-J].

4.6.7 ERR02-J. Prevent Exceptions While Logging Data

```
try {
 // ...
} catch (SecurityException se) {
 System.err.println(se);
 // Recover from exception
}
```

Writing exceptions to the standard error stream is inadequate for logging purposes because the standard error stream may get crashed and the trust level of the standard error stream may be deficient[CERT/ERR02-J].

4.6.8 ERR04-J. Do Not Complete Abruptly from a Finally Block

```
class TryFinally {
 private static boolean doLogic() {
  try {
   throw new IllegalStateException();
  } finally {
   System.out.println("logic done");
   return true;
  }
 }
}
```

The finally block may complete abruptly because of a return statement in the block[CERT/ERR04-J].

4.6.9 ERR05-J. Do Not Let Checked Exceptions Escape from a Finally Block

```
public class Operation {
 public static void doOperation(String some_file) {
  // ... Code to check or set character encoding ...
  try {
   BufferedReader reader =
    new BufferedReader(new FileReader(some_file));
   try {
    // Do operations
   } finally {
    reader.close();
    // ... Other cleanup code ...
   }
  } catch (IOException x) {
   // Forward to handler
  }
 }
}
```

The close() method can throw an IOException. If thrown, it would prevent the execution of any subsequent cleanup statements[CERT/ERR05-J].

4.6.10 ERR07-J. Do Not Throw RuntimeException, Exception, or Throwable

```
boolean isCapitalized(String s) {
 if (s == null) {
  throw new RuntimeException("Null String");
 }
 if (s.equals("")) {
  return true;
 }
 String first = s.substring(0, 1);
 String rest = s.substring(1);
 return (first.equals(first.toUpperCase()) &&
  rest.equals(rest.toLowerCase()));
}
```

4.6.11 ERR08-J. Do Not Catch NullPointerException or Any of Its Ancestors

```
boolean isName(String s) {
 try {
  String names[] = s.split(" ");

  if (names.length != 2) {
   return false;
  }
  return (isCapitalized(names[0]) && isCapitalized(names[1]));
 } catch (NullPointerException e) {
  return false;
 }
}
```

4.6.12 FIO02-J. Detect and Handle File-related Errors

```
public class FileClass {
 public fileDelete() {
  File file = new File(args[0]);
  file.delete();
 }
}
```

It attempts to delete a specified file but it don't give any message that the operation is success or not[CERT/FIO02-J].

4.6.13 MET09-J. Classes That Define an Equals() Method Must Also Define a Hashcode() Method

```
public final class CreditCard {
 private final int number;
 public CreditCard(int number) {
  this.number = number;
 }
 public boolean equals(Object o) {
  if (o == this) {
   return true;
  }
  if (!(o instanceof CreditCard)) {
   return false;
  }
  CreditCard cc = (CreditCard) o;
  return cc.number == number;
 }
 public static void main(String[] args) {
  Map < CreditCard, String > m = new HashMap < CreditCard, String > ();
  m.put(new CreditCard(100), "4111111111111111");
  System.out.println(m.get(new CreditCard(100)));
 }
}
```

The expected retrieved value is 4111111111111111; the actual retrieved value is null because CreditCard class overrides the equals() method but fails to override the hashCode() method. So, the default hashCode() method returns a different value for each object[CERT/MET09-J].

4.6.14 MSC00-J. Use SSLSocket Rather than Socket for Secure Data Exchange

```
class EchoServer {
 public static void main
  (String[] args) throws IOException {
  ServerSocket serverSocket = null;
  try {
   serverSocket = new ServerSocket(9999);
   Socket socket = serverSocket.accept();
   PrintWriter out = new PrintWriter
    (socket.getOutputStream(), true);
   BufferedReader in = new BufferedReader(
    new InputStreamReader(socket.getInputStream
    ()));
   String inputLine;
   while
   ((inputLine = in .readLine()) != null) {
    System.out.println(inputLine);
    out.println(inputLine);
   }
  } finally {
   if (serverSocket != null) {
    try {
     serverSocket.close();
    } catch
    (IOException x) {
     // Handle error
    }
   }
  }
 }
}
```

It uses regular sockets for a server application. It is vulnerable if a sensitive information is in transit[CERT/MSC00-J].

4.6.15 MSC01-J. Do Not Use an Empty Infinite Loop

```
public class loop {
 public int nop() {
  while (true) {}
 }
}
```

In the above example an unused task that continuously performs the while loop without executing any code inside it[CERT/MSC01-J].

4.6.16 MSC02-J. Generate Strong Random Numbers

```
import java.util.Random;
public class RandomNumber
public static void main(String args[]) {
 Random number = new Random(123 L);
 for (int i = 0; i < 20; i++) {
  int n = number.nextInt(21);
  System.out.println(n);
 }
}
```

The code produces the same sequence of numbers for the same seed value. So, the sequence of numbers is predictable[CERT/MSC02-J].

5

Implementation and Results

This chapter contains the implementation details of the CERT rules and the result we obtained with our tool. The details about the runtime data and Java code used for the implementation of the tool is described in Appendix A and Appendix B respectively.

5.1 List of CERT Rules Implemented

The following rules are implemented in the tool currently:

1. DCL00-J. Prevent class initialization cycles
2. DCL01-J. Do not reuse public identifiers from the Java Standard Library
3. DCL02-J. Do not modify the collection's elements during an enhanced for statement
4. EXP02-J. Do not use the Object.equals() method to compare two arrays
5. ERR02-J. Prevent exceptions while logging data
6. ERR04-J. Do not complete abruptly from a finally block
7. ERR05-J. Do not let checked exceptions escape from a finally block
8. ERR07-J. Do not throw RuntimeException, Exception, or Throwable
9. ERR08-J. Do not catch NullPointerException or any of its ancestors
10. ERR02-J. Prevent exceptions while logging data
11. FIO02-J. Detect and handle file-related errors
12. FIO03-J. Remove temporary files before termination

13. LCK01-J. Do not synchronize on objects that may be reused
14. MET00-J. Validate method arguments
15. MET09-J. Classes that define an equals() method must also define a hashCode() method
16. MSC00-J. Use SSLSocket rather than Socket for secure data exchange
17. MSC01-J. Do not use an empty infinite loop
18. MSC02-J. Generate strong random numbers
19. MSC06-J. Do not modify the underlying collection when an iteration is in progress
20. NUM07-J. Do not attempt comparisons with NaN
21. NUM09-J. Do not use floating-point variables as loop counters
22. NUM10-J. Do not construct BigDecimal objects from floating-point literals
23. OBJ09-J. Compare classes and not class names
24. OBJ10-J. Do not use public static nonfinal fields
25. SER05-J. Do not serialize instances of inner classes

5.2 List of Instances of Non-Compliance

This section contains the list of violations identified by the tool. The full collection of all instances of non-compliance of the CERT rules we implemented appears as Appendix A. Each item in the list includes the pathname (relative to the root of AOSP source code tree), the range of line numbers, the CERT rule identifier and a brief explanation of the non-compliance and a fix.

5.2.1 ERR05-J. Do Not Let Checked Exceptions Escape From a Finally Block

Methods invoked from within a finally block can throw an exception. Failure to catch and handle such exceptions results in the abrupt termination of the entire try block. Abrupt termination causes any exception thrown in the try block to be lost, preventing any possible recovery method from handling that specific problem. If any method invocation present inside the finally block without a try-catch block will violate this rule, based on this logic we wrote the program and more than 200 true positive violations reported in AOSP, one example from binder is `frameworks/base/core/`

`java/android/accessibilityservice/AccessibilityServiceInfo.java` line number 520. The full violations list is present in Appendix A (see Section 9.2) and the source code used to detect this rule is present in Appendix B (see Section 10.9).

5.2.2 ERR07-J. Do Not Throw RuntimeException, Exception, or Throwable

If any class object is created for RuntimeException, Exception, or Throwable inside a throw block this violation will be present, based on this logic we wrote the program and more than 350 true positive violations reported in AOSP. To detect this violation we first convert the source code to AST and visited the throw node and search any class object is created for RuntimeException, Exception, or Throwable. One example from AOSP is file `frameworks/base/core/java/android/net/NetworkPolicyManager.java` line number 310. The full violations list is present in Appendix A (see Section 9.1) and the source code used to detect this rule is present in Appendix B (see Section 10.10).

5.2.3 ERR08-J. Do Not Catch Nullpointerexception or Any of Its Ancestors

A NullPointerException exception thrown at runtime shows the presence of a null pointer dereference that must be fixed in the application code. If exception type in the catch block is NullPointerException then it is non-compliant to ERR08-J. Based on this logic there are 18 true positive violations reported in AOSP binder part. One example from AOSP is file `frameworks/base/core/java/android/net/NetworkUtils.java` line number 379. The full violations list is present in Appendix A (see Section 9.4) and the source code used to detect this rule is present in Appendix B (see Section 10.11).

5.2.4 FIO02-J. Detect and Handle File-related Errors

Programs that ignore the return values from file operations often fail to detect that those operations have failed. This violation may lead to information disclosure, memory wastage, etc. There are 18 true positive violations reported in AOSP binder part. One example from AOSP is file `frameworks/base/core/java/android/app/backup/WallpaperBackupHelper.java` line number 169 File object named “f” tried to use delete() without checking file related errors. The full violations list is present in Appendix A (see Section 9.7) and the source code used to detect this rule is present in Appendix B (see Section 10.12).

5.2.5 MSC00-J. Use SSLSocket Rather than Socket for Secure Data Exchange

Programs must use the javax.net.ssl.SSLSocket class rather than the java.net.Socket class when transferring sensitive data over insecure channels. If the Socket class is used in the program then it may noncompliance to MSC01-J. Only one violation is detected in the binder on the following file `frameworks/base/core/java/android/net/Network.java`, line number 179. There is an exceptional case for MSC01-J, if the data is not sensitive or encrypted we can use Socket class itself. Human intervention is needed to verify such exceptions. The source code used to detect this rule is present in Appendix B (see Section 10.14).

5.2.6 MSC02-J. Generate Strong Random Numbers

Two instances of the java.util.Random class that are created using the same seed will generate identical sequences of numbers. An attacker can learn the value of the seed by performing some observation on the vulnerable target. If the Random class is used in the program it may have the MSC02-J violation. There are 4 such kind of violations reported in binder. The full violations list is present in Appendix A (see Section 9.3) and the source code used to detect this rule is present in Appendix B (see Section 10.16)

5.3 AST Traversal Example

The tool first converted the source code to AST with the help of eclipse JDT and later it did a pattern matching on the AST to find a specific violations. Consider the implementation code of OBJ10-J

```
public boolean visit(FieldDeclaration node) {
 int flag_pub = 0;
 int flag_sta = 0;
 int flag_fin = 0;
 for (int i = 0; i < node.modifiers().size(); i++) {

  if (node.modifiers().get(i).toString().equals("public")) {
   flag_pub = 1;

  }
  if (node.modifiers().get(i).toString().equals("static")) {
   flag_sta = 1;
  }
  if (node.modifiers().get(i).toString().equals("final")) {
   flag_fin = 1;

  }
 }
 if (flag_fin == 0 && flag_pub == 1 && flag_sta == 1) {
  PrintError(OBJ10 - J, Lineno);
  Main.obj10j++;
 }
 return true; //
}
```

The rule says do not use public static nonfinal fields. So the program first visits the FieldDeclaration node and get all the modifiers using the method modifiers(). If the “public” “static” “final” is present then corresponding flag values “flag_pub”, “flag_sta”, “flag_fin” will set to 1. If a condition like flag_pub=1, flag_sta=1, flag_fin=0 indicates public static nonfinal field present and it triggers the error message. Appendix B contains the AST traversal code files for all the rules we implemented (see Section 5.1).

6 Related Work

There are a few tools and coding standards that help to verify the given source code and identify potential violations.

6.1 Secure Coding Guide Lines

6.1.1 Oracle Java SE Security

One of the main design concerns for the Java platform is to provide a secure environment with the help of its own unique set of security challenges. It cannot defend against implementation flaws that occur in the source code. These bugs may accidentally open the very holes that the security architecture was designed to contain, including access to files, printers, webcams, microphones, etc. These bugs can probably be used to turn the machine into a zombie computer, steal confidential data, user credentials, spy through attached devices, limit the serviceable operation of the machine and many other malicious activities[Oracle 2014].

Java security technology includes a large set of APIs, tools, and implementations of commonly used security algorithms, mechanisms, and protocols. It is for a wide range of areas, including cryptography, public key infrastructure, secure communication, authentication, and access control. There are secure coding guidelines also recommended. The main categories are fundamentals, denial of service, confidential information, injection and inclusion, accessibility and extensibility, input validation, mutability, object construction, serialization and deserialization, access control. Out of these ten the fundamental security deals the basic guidelines and it is as follows:

1. Prefer to have obviously no flaws rather than no obvious flaws

2. Design APIs to avoid security concerns

3. Restrict privileges
4. Establish trust boundaries
5. Minimize the number of permission checks
6. Encapsulate
7. Document security-related information

6.1.2 CERT Secure Coding Standard

The CERT Secure Coding Standard[Long et al. 2011] provides rules and recommendations (collectively called guidelines) for secure coding in the particular programming language. The goal of these rules and recommendations is to develop safe, reliable, and secure systems. Conformance to the coding rules defined in this standard are necessary (but not sufficient) to ensure the safety, reliability, and security of software systems developed in the programming language. CERT developed coding standards for Android, C, C++, Java and Perl.

6.1.2.1 Rule Identifiers

Each rule has a unique identifier, consisting of three parts:

- **A three-letter mnemonic**, representing the section of the standard, is used to group similar rules and make them easier to find. *eg: PRE: for preprocessor*
- **A two-digit numeric value** in the range of 00 to 99, which ensures each rule has a unique identifier *eg: PRE30.*
- **The letter J/C**, which indicates that this is a Java language/C language rule and is included to prevent ambiguity with similar rules in CERT secure coding standards for other languages. *eg: PRE30-C.*. Finally, the rule looks like, *eg: PRE30-C - Do not create a universal character name through concatenation.*

6.1.2.2 Priority and Levels

Each rule has an assigned priority. Priorities are assigned using a metric based on Failure Mode, Effects, and Criticality Analysis. Three values are assigned for each rule on a scale of 1 to 3 for:

Severity How serious are the consequences of the rule being ignored:

- 1 = low The effect is denial-of-service attack, abnormal termination.

- 2 = medium The effect is data integrity violation, unintentional information disclosure.
- 3 = high The effect is run arbitrary code, privilege escalation.

Likelihood The probability that a flaw introduced by violating the rule could lead to an exploitable vulnerability:

- 1 = unlikely
- 2 = probable
- 3 = likely

Remediation cost How expensive is it to remediate existing code to comply with the rule:

- 1 = high The effect is manual detection and correction.
- 2 = medium The effect is automatic detection and manual correction.
- 3 = low The effect is automatic detection and correction.

The three values are multiplied together for each rule. This product provides a measure to prioritize the rules. These products range from 1 to 27. Rules with a priority in the range of 1 to 4 are level 3 rules, 6 to 9 are level 2, and 12 to 27 are level 1. As a result, it is possible to claim level 1, level 2, or complete compliance (level 3) with a standard by implementing all rules in a level, as shown in Figure 6.1.

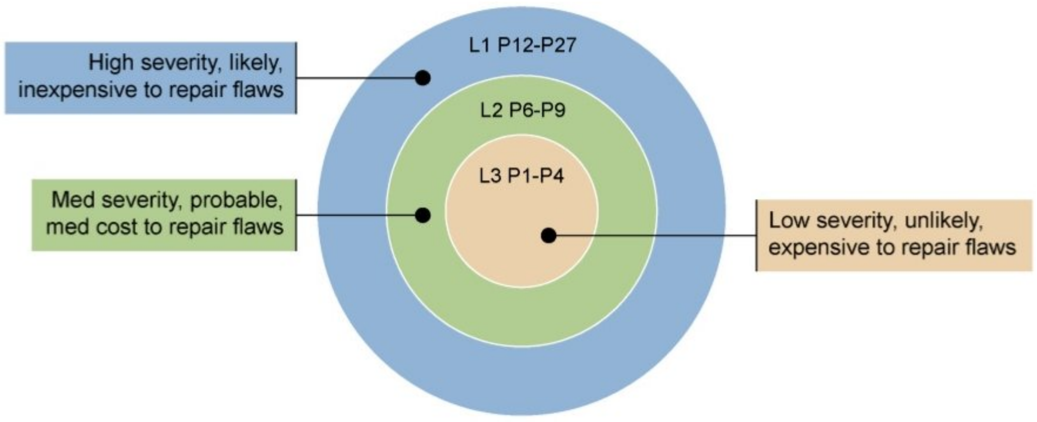

High severity, likely, inexpensive to repair flaws
L1 P12-P27
L2 P6-P9
L3 P1-P4
Med severity, probable, med cost to repair flaws
Low severity, unlikely, expensive to repair flaws

Figure 6.1: Levels and priority ranges

[Long et al. 2011]

6.1.2.3 Rules and Recommendations

Rules are normalized ones. It is mandatory to follow rules.Rules must meet the following criteria:

1. Violation of the guideline is likely to result in a defect that may adversely affect the safety, reliability, or security of a system.

2. The guideline does not rely on source code annotations or assumptions. Rules are identified by the label rule.

Recommendations are not normalized. Recommendations are suggestions for improving code quality. It is not mandatory to follow recommendations. Guidelines are defined to be recommendations when all of the following conditions are met:

1. Application of a guideline is likely to improve the safety, reliability, or security of software systems.

2. One or more of the requirements necessary for a guideline to be considered a rule cannot be met.

Recommendations are identified by the label recommendation.

6.1.2.4 CERT Android Secure Coding Standard

The Android standards are classified into C, Java, Android only rules. Out of this three Android rules are under development stage, so these are incomplete or contain errors. There are twenty eight rules listed in CERT website related to Android, but these are not mature. They haven't published any report or book form as official releases. But some of the standardized and officially released CERT C and Java secure coding rules are applicable to developing Android applications. There are 17 Android C rules and 10 recommendations. In the case of Android Java 160 rules and 46 recommendations are available. Table 6.1.2.4 and table 6.1.2.4 give an idea about Android C and Java rules.

Rule Id	**Name**	**Rules**	**Recommendations**
01	Preprocessor (PRE)	-	-
02	Declarations and Initialization (DCL)	01	-
03	Expressions (EXP)	01	-
04	Integers (INT)	01	-
05	Floating Point (FLP)	03	04
06	Arrays (ARR)	-	-
07	Characters and Strings (STR)	02	01
08	Memory Management (MEM)	01	-
09	Input Output (FIO)	03	-
10	Environment (ENV)	-	-
11	Signals (SIG)	04	03
12	Error Handling (ERR)	-	-
13	Application Programming Interfaces (API)	-	02
14	Concurrency (CON)	-	-
48	Miscellaneous (MSC)	01	-
50	POSIX (POS)	-	-

Table 6.1: Categories in the CERT Android C secure coding standard

Rule Id	Name	Rules	Recommendations
00	Input Validation and Data Sanitization (IDS)	4	6
01	Declarations and Initialization (DCL)	03	-
02	Expressions (EXP)	02	02
03	Numeric Types and Operations (NUM)	9	04
04	Characters and Strings (STR)	1	-
05	Object Orientation (OBJ)	10	-
06	Methods (MET)	10	03
07	Exceptional Behavior (ERR)	9	01
08	Visibility and Atomicity (VNA)	06	-
09	Locking (LCK)	8	4
10	Thread APIs (THI)	01	-
11	Thread Pools (TPS)	-	-
12	Thread-Safety Miscellaneous (TSM)	-	03
13	Input Output (FIO)	02	-
14	Serialization (SER)	11	01
15	Platform Security (SEC)	01	04
16	Runtime Environment (ENV)	-	02
17	Java Native Interface (JNI)	-	-
49	Miscellaneous (MSC)	07	01

Table 6.2: Categories in the CERT Android Java secure coding standard

6.2 Static Analysis for Secure Coding

The most efficient way to gain software security is to analyze past security errors and prevent them from happening in the future. On effective solution is by documenting the security errors present in past and do a code review with the help of static analysis technique. The term static analysis refers to any process for evaluating code without running it. Static analysis is powerful because it allows for the quick consideration of many possibilities. A static analysis tool can explore a large number of scenarios without having to go through all the computations necessary to execute the code for all the scenarios. Code review should be part of the software security process. When used as part of code review, static analysis tools can help classify best practices, catch common mistakes, and generally make the security process more efficient and consistent. At a high level, the process consists of four steps: defining goals, running tools, reviewing the code, and making fixes. There are some practical challenges for static analysis tools include the following. All static analysis tools produce at least some false positives or some false negatives but most produce both. For security purposes, false negatives are more troublesome than false positives, although too many false positives can lead a reviewer to overlook true positives[Chess 2007]. The figure 6.2 will show how to perform a code review with a tool.

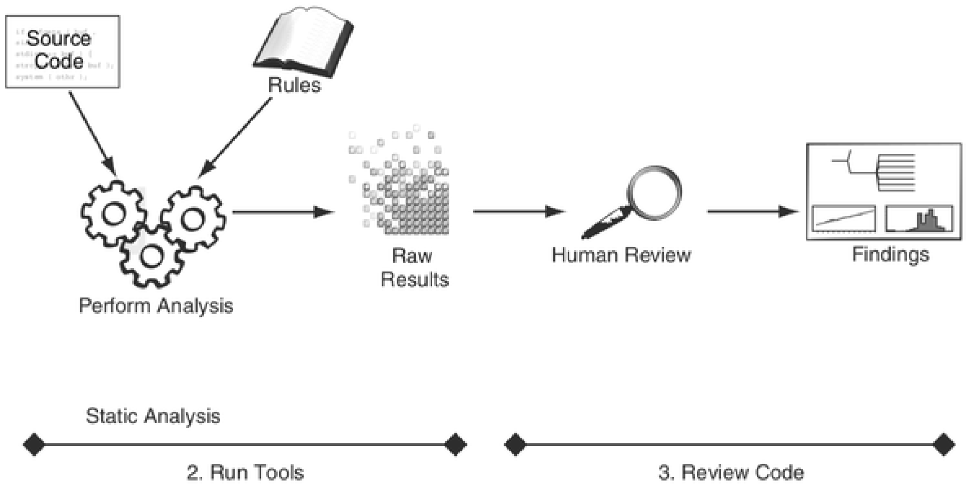


Figure 6.2: Code review with a tool
[Chess 2007]

6.2.1 Queries on Java Programs

Java source code files are transformed into ASTs and can be further deconstructed into a collection of logical facts that interrelate methods, expressions, variables, and so forth. On this facts repository of gigabytes, queries can be executed using logic programming languages.

6.2.1.1 JTransformer

Code query technologies is an important object of software analysis like found in software architecture analysis, applying consistency checks, enforcing coding conventions and reverse engineering. For custom analyses and transformations of Java source code JTransformer is a powerful tool. It is available with GUI and a plug-in for the Eclipse JDT. It will handle full source code of a Java project including the comments, declarations of program elements contained in class files, methods, statements, loops, etc. One of the important API that they are providing is Java AST and Program Element Facts (PEF). JTransformer represents the Java AST and information about source code, directories by Prolog facts called Program Element Facts (PEFs). The PEF signs end with a capital S , example sourceFolderS, projectS, fileS, etc. All other PEF types represent language-specific components of the Java AST which end with T[Kniesel et al. 2007].

One example of Jtransformer AST[iai.uni bonn.de 2011] is:

Sample Java Source

```
package packaged.de.tests;
public class tests {
...
}
```

Its PEF Representation

```
packageT(#Package, 'packaged.de.tests'),
compilationUnitT(#CompilationUnit, #Package, ..., [], #Class),
classT(#Class, #CompilationUnit, 'tests', [#Children]),
```

AST Specification

```
ast_node_def('Java',packageT,[
ast_arg(id,  mult(1,1,no ), id, [id]), % <-- convention!!!
ast_arg(name, mult(1,1,no ), attr, [atom])
]).
```

6.2.1.2 Eclipse Java Development Tools(JDT)

The Abstract Syntax Tree is one of the base frameworks for many powerful tools of the Eclipse IDE, including refactoring, Quick Fix, and Quick Assist[Kuhn 2006] The Abstract Syntax Tree maps plain Java source code in a tree form. This tree is more convenient and reliable to analyze and modify programmatically than text-based source. Eclipse JDT is a nice plug in which converts source code to AST. It is defined in the org.eclipse.jdt.core plug-in.

The main package for the AST is the org.eclipse.jdt.core.dom package and is located in the org.eclipse.jdt.core plug-in. Each Java source file is represented as a subclass of the ASTNode class. Each specific AST node provides specific information about the object it represents. For example It has MethodDeclaration (for methods), Expressions (Expressions), IfStatement (For if-else) VariableDeclarationFragment (for variable declarations) and SimpleName (for any string which is not a Java keyword), etc. First, the AST is created based on an ICompilationUnit from the plugin, then visit each node using visitor pattern algorithm. A typical workflow of an application using AST is as follows:

1. Java source: To start off, you provide some source code to parse. This source code can be supplied as a Java file in your project or directly as char arraythat contains Java source.
2. Parse: Most of the time, an AST is not created from scratch. This is done using theASTParser. It processes whole Java files as well as portions of Java code.
3. The Abstract Syntax Tree: It is a tree model that entirely represents the source you provided for parsing. The parser also generates and includes additional symbol resolved information called bindings.
4. Manipulating the AST: For some operations like refactoring, Quick Fix and Quick Assist AST should modify, this can be done in two ways:
   (a) By directly modifying the AST.
   (b) By noting the modifications in a separate protocol. This protocol is handled by an instance ofASTRewrite.
5. Writing changes back: Once changes have been tracked, either by usingASTRewriteor by modifying the tree nodes directly, these changes can be written back into Java source code. Therefore, aTextEditobject has to be created. Here we leave the code related area of the AST and enter a text-based environment. TheTextEditobject contains character based modification information. It is part of theorg.eclipse.textplug-in.
6. IDocument: Is a wrapper for the source code of parsing and is needed for writing changes back

6.2.1.3 JavaParser- A Case Study

A JavaParser is a tool to parse Java 8 code with AST generation and visitor support. The AST records the source code structure, Javadoc and comments. It is also possible to change the AST nodes or create new ones to modify the source code. It is more convenient and reliable to analyze and modify programmatically than text-based source. The main features are Light weight, Easy to use, Modifiable AST, Create AST from scratch, etc. This parser was created using javacc (the java compiler compiler). All the nodes of the AST, visitors and other features was coded manually using the Eclipse IDE.

From JavaParser other projects have been derived. Walkmod[?], a tool to automatically correct violations of code conventions is one example. /* Yet to finish */

6.2.1.4 JetBrains MPS

JetBrains MPS[Pech et al. 2013] is an integrated environment for language engineering. It provides language designers to define new programming languages, both general-purpose and domain-specific. Since MPS supports the concept of projectional editing, non-textual and non-parseable syntactic forms are possible, including tables or mathematical symbols. MPS offers a set of DSLs for defining various aspects of languages which include the structure, editor, type system, the generator as well as support for sophisticated IDE functionality, such as code completion, intentions, refactorings, debugger and dataflow analysis. The first step is defining a new language specify the structure. In MPS structure is defined using AST. The types of AST nodes is represented by concepts in MPS and define the properties, methods, children and references that instance nodes may have. MPS offers three language aspects to define abstract syntax: the structure aspect defines the concepts, their properties and relationships, constraints restrict the allowed set of values for properties and references, and the behavior associates methods with concepts. The second step is defining the editor for the concepts. This reflects the projectional nature of MPS. Since the code in MPS is never represented as plain text, MPS languages are never parsed and thus no grammar is required. This enables the use of non-parseable notations, such as tables or mathematical symbols. Instead of a parser, it defines editors for language concepts, a visual representation of AST nodes. The editor offers many of its fundamental features out-of-the-box, the remainder can be implemented by the language designer through appropriate language aspects. The next step is the definition of the type system. MPS have a type system engine that is capable of assessing type system rules and to verify the correctness of types. The next is code generation, consists of two phases. Phase one uses a template-based model-to-model transformation engine to reduce the program code into the target language, based on reduction rules specified in the generator. The second phase uses text generators to convert that final model into regular program text that can be fed into a compiler.

6.2.1.5 FaceBook Infer

Facebook Infer[FbInfer] is a static analysis tool for Objective-C, Java, or C code. It produces a list of potential bugs present in the code. Anyone can use Infer to catch critical bugs before they have released their product, and help prevent crashes or poor performance. The two phases of an Infer run are the capture phase and the analysis phase. In capture phase, compilation commands are captured by Infer and translate the file into Infers own internal intermediate language. This translation is similar to compilation with a compilation command infer – javac File.java for java files and infer – clang -c file.c for C files. Infer stores the intermediate files in the results directory. The default is created in the folder where the infer command is invoked and is called infer-out/.

In the analysis phase, the files in infer-out/ are analyzed by Infer. Infer analyzes each function and method separately. If it finds an error when analyzing a method or function, it stops there for that method or function, but will continue the analysis of other methods and functions. The errors will be displayed in the standard output and also in a file infer-out/bugs.txt. It filters the bugs and shows the ones that are most likely to be real. In the results directory (infer-out/), however, it also saves a file report.csv that contains all the errors, warnings, and infos reported by Infer in CSV format.

6.2.2 Security Analysis Tools

6.2.2.1 Rose Compiler

Rose[Quinlan 2000] was developed by Lawrence Livermore National Labs (LLNL). It will analyzes program source code, Produces Abstract Syntax Tree (AST), Can then be used for static analysis. There are a lot of dependencies and usage of old binaries makes rose a bad experience for the programmer. So they pre-installed all the packages in a virtual machine and published this image named rosebud. In this image rosechecker also installed. The CERT Division's rosecheckers tool performs static analysis on C/C++ source files. It is designed to enforce the rules in the CERT C Coding standard. Rosecheckers finds some C coding errors that other static analysis tools do not. However, it does not do a comprehensive test for secure and correct C coding, and it is only a prototype, so it cannot be used alone to fully analyze code security. Rosecheckers can be run on a C or C++ file. The Rosecheckers program displays the file's violations of the secure coding rules that it is programmed to check for. Rosecheckers takes the same arguments as gcc, so code that contains special flags that must be passed to the compiler can be passed to rosecheckers in the same manner as gcc. For compile and CERT violation detection the following command is defined:

```
rosechecker example.c
```

6.2.2.2 Coccinelle

Coccinelle [Olesen et al. 2014], explains a prototype tool for automated CERT C Secure Coding Standard compliance checking. The tool is based on the open source program analysis and program transformation tool Coccinelle that has been successfully used to find bugs in the program written in C languages like Linux kernel, the OpenSSL, and other open source infrastructure software. Coccinelle is using a domain-specific language, called the Semantic Patch Language (SmPL), as well as OCaml and Python. The scripts, called semantic patches, specify search patterns partly based on syntax and partly on the control flow of a program. This makes Coccinelle easily adaptable to new classes of errors and new code bases with distinct API usage and code-style requirements. Coccinelle performs automated CERT C Secure Coding Standard compliance checking, it does not perform program analysis like data-flow analysis or range analysis. For the purposes of program certification and compliance checking such analyses are essential, both to ensure soundness of the certification and to improve precision of the tool. For this reason they are currently working on integrating the Clang Static Analyzer with Coccinelle in order to enable Coccinelle to use the analysis (and other) information found by Clang. The Clang Static Analyzer is part of the C front-end for the LLVM project.2 In addition to classic compiler support, it also provides general support for program analysis, using a monotone framework, and provides a framework for checking source code for (security) bugs. The emphasis in the source code checkers of the Clang project is on minimizing false positives (reporting errors that are not really errors) and thus it is likely to miss some real error cases.

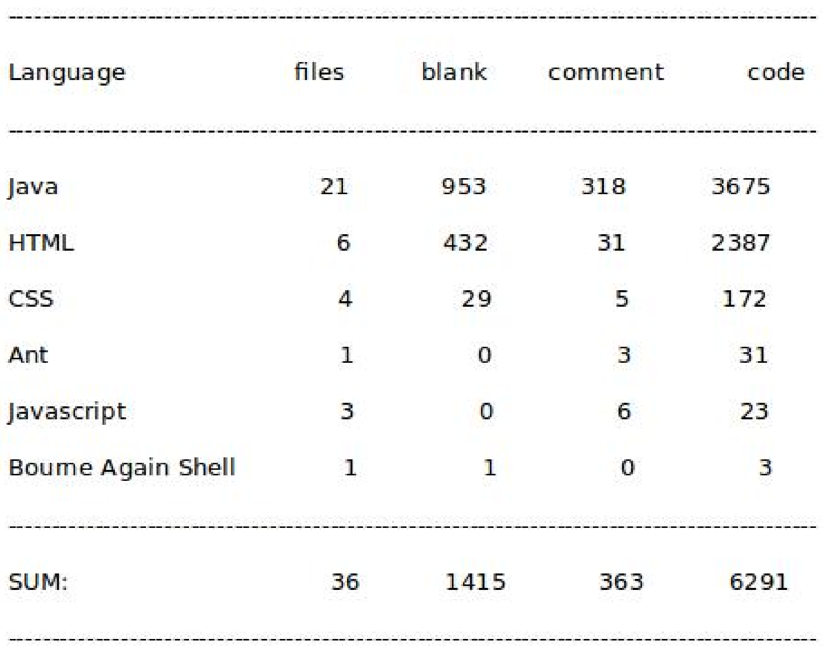
7

Evaluation

This chapter contains the evaluation details of our work like loc, auditing details, running time, false positives etc.

7.1 Development Details

The tool named `Rast` and an executable binary with documentation is ready to download at `https://github.com/rahulbpkl/rast/`. The tool developed in Java and the description is written in HTML. The full source code of the binary is available at the following link `https://github.com/rahulbpkl/Android-Java-Audit-Draft-1`. The total size of the binary including all the configuration file is 15MB. For all the background work, testing, coding we took nearly five months with an average 48 hours per week. To scan 438626 lines of code in an i3 processor 8GB RAM laptop it took nearly 6 minutes. The SLOC is of in Figure 7.1



```
-------------------------------------------------------------------------------
Language                 files      blank     comment       code
-------------------------------------------------------------------------------
Java                        21        953         318       3675
HTML                         6        432          31       2387
CSS                          4         29           5        172
Ant                          1          0           3         31
Javascript                   3          0           6         23
Boume Again Shell            1          1           0          3
-------------------------------------------------------------------------------
SUM:                        36       1415         363       6291
-------------------------------------------------------------------------------
```

Figure 7.1: SLOC of RAST

7.2 Auditing Details

The AOSP is big in size so scanning and analyze full Java code will take a lot of efforts. In this thesis, we addressed violations of Binder part present in the location `frameworks/base/core`. In this other than actual program codes some test code and debug code are also present. Our program has the feature to exclude some directories and we excluded all unwanted directories. Total 796 files with 438626 lines of code were scanned and out of this 744 lines in 130 files having the CERT violations.

7.3 False Positives

A false positive is a problem reported in a program when no problem actually exists. Almost all the automation tools have the false positive problem. There are 2 MSC02-J[CERT/MSC02-J] false positive problems reported while auditing `frameworks/base/core`. According to MSC02-J generate strong random numbers but CERT documented some exceptions for this rule. Our tool detected a violation but it is actually present in the exceptional case. Eliminating such kind of false positive is out of the scope of this thesis.

8

Conclusion and Future Work

8.1 Conclusion

Secure coding standards plays an essential role in the software industry, and the violations may lead to a severe security flaw. The analysis shows still there are lots of coding standard violations. In this thesis, we addressed the violations present in a portion of Binder. There are more than five hundred violations reported in this portion. It indicates that Android doesn't comply with CERT standard. The tool developed for detecting the code standard violation with proper documentation is ready to download at `https://github.com/rahulbpkl/rast/` and the full source code is available at the following link `https://github.com/rahulbpkl/Android-Java-Audit-Draft-1`.

8.2 Future Work

8.2.1 User Interface

Currently, we implemented a command line tool with arguments. Most o the programmers always use IDE for development. Even Android Studio also have a UI. One of the future work is to create a UI for the tool and it will be either a new one or a plug-in for the existing IDE like JetBrains, Eclipse. It will help the programmer to write compliant code while typing itself.

8.2.2 Implement Greater Number of Rules

There are 84 rules for Android Java and currently, we implemented only 25 rules. There are some rules which cannot be implemented by tools, human intervention is needed and other rules are not possible with AST concept. Implementing more rules and maintaining the scalability is one of the future work.

8.2.3 Build a New ROM

Finding the code violations doesn't make any use to the existing OS and it may help a clever attacker to penetrate into the system. Someone has to fix the error and build a more secure ROM. The correction and building ROM is another research project. Moreover the verification of security before and after is also another important thing.

9

Appendix A List of Violations

This chapter contains the full collection of all instances of non-compliance of the CERT rule present in the binder. The collected information is arranged in the following way "filename" -"line number"

9.1 ERR07-J. Do Not Throw RuntimeException, Exception, or Throwable

Throwing a RuntimeException may lead to complex errors, for example, a caller cannot examine the exception to determine why it was thrown and consequently cannot attempt recovery. To fix such kind of exception use specific exceptional condition for example NullPointerException, IOException, etc. The full collection of all instances of non-compliance of the ERR07-J present in the binder is listed below:

1. frameworks/base/core/java/android/net/nsd/NsdManager.java - 439
2. frameworks/base/core/java/android/net/NetworkPolicyManager.java - 310.
3. frameworks/base/core/java/android/net/NetworkUtils.java - 317,347
4. frameworks/base/core/java/android/net/TrafficStats.java - 272, 615, 628.
5. frameworks/base/core/java/android/net/LocalSocketImpl.java - 428
6. frameworks/base/core/java/android/accounts/AccountManager.java - 337, 366, 386, 408, 432, 457, 474, 492,530, 550, 715, 740, 809, 980, 1008, 1040, 1070, 1099, 1128, 1158, 1556, 1607, 1621, 1943, 2116
7. frameworks/base/core/java/android/animation/AnimatorInflater.java - 713.

8. frameworks/base/core/java/android/content/res/AssetManager.java - 311, 327, 403, 424, 483, 504.
9. frameworks/base/core/java/android/content/res/TypedArray.java - 84, 97, 115, 128, 149, 167, 186, 205, 225, 256, 278, 300, 320, 339, 89, 376, 401, 425, 476, 507, 553, 600, 648, 690, 729, 766, 806, 834, 861, 892, 914, 930, 950, 971, 995, 1013, 1027, 1062, 1112.
10. frameworks/base/core/java/android/content/res/XmlBlock.java - 124, 156, 191, 422.
11. frameworks/base/core/java/android/database/BulkCursorToCursorAdaptor.java - 172.
12. frameworks/base/core/java/android/ddm/DdmHandleHeap.java - 103.
13. frameworks/base/core/java/android/ddm/DdmHandleHello.java - 95.
14. frameworks/base/core/java/android/ddm/DdmHandleNativeHeap.java - 66.
15. frameworks/base/core/java/android/ddm/DdmHandleProfiling.java - 94.
16. frameworks/base/core/java/android/ddm/DdmHandleThread.java - 79.
17. frameworks/base/core/java/android/ddm/DdmHandleViewDebug.java - 164.
18. frameworks/base/core/java/com/android/internal/app/ChooserActivity.java - 632, 649.
19. frameworks/base/core/java/com/android/internal/app/ProcessStats.java - 1448.
20. frameworks/base/core/java/com/android/internal/inputmethod/InputMethodUtils.java - 85, 99.
21. frameworks/base/core/java/com/android/internal/os/Zygote.java - 167.
22. frameworks/base/core/java/com/android/internal/os/ZygoteInit.java - 291, 313, 484, 536, 582, 587, 679, 728, 736.
23. frameworks/base/core/java/com/android/internal/os/PowerProfile.java - 247, 249.
24. frameworks/base/core/java/com/android/internal/policy/PhoneWindow.java - 3915, 4140, 4187.
25. frameworks/base/core/java/com/android/internal/util/StateMachine.java - 798, 1156.
26. frameworks/base/core/java/com/android/internal/util/WithFramework.java - 40.
27. frameworks/base/core/java/com/android/internal/util/XmlUtils.java - 710.
28. frameworks/base/core/java/com/android/internal/util/HexDump.java - 149.

29. frameworks/base/core/java/com/android/internal/widget/AutoScrollHelper.java - 818.
30. frameworks/base/core/java/com/android/internal/widget/ExploreByTouchHelper.java - 298, 401, 407, 413, 417.
31. frameworks/base/core/java/com/android/internal/widget/LockPatternUtils.java - 1243.
32. frameworks/base/core/java/com/android/internal/widget/TextProgressBar.java - 116, 134.

9.2 ERR05-J. Do Not Let Checked Exceptions Escape from a Finally Block

Methods invoked from within a finally block may throw an exception. Failure to catch and handle such exceptions results in the abrupt termination of the entire try block. Abrupt termination causes any exception thrown in the try block to be lost, preventing any possible recovery method from handling that specific problem. To fix this problem enclose the method invocation with a try-catch block. The full collection of all instances of non-compliance of the ERR05-J present in the binder is listed below:

1. frameworks/base/core/java/android/net/SntpClient.java
2. frameworks/base/core/java/android/accessibilityservice/AccessibilityServiceInfo.java - 520.
3. frameworks/base/core/java/android/animation/AnimatorInflater.java - 150.
4. frameworks/base/core/java/android/animation/AnimatorInflater.java - 197.
5. frameworks/base/core/java/android/app/ActivityThread.java - 980, 2661, 2737, 2964, 2980, 2996, 4300, 4323, 4716.
6. frameworks/base/core/java/android/app/DownloadManager.java - 1038, 1066, 1090.
7. frameworks/base/core/java/android/app/SharedPreferencesImpl.java - 128.
8. frameworks/base/core/java/android/app/UiAutomation.java - 555, 937.
9. frameworks/base/core/java/android/app/UiAutomationConnection.java - 121,143, 159, 178, 197, 212, 229, 245, 261, 301.
10. frameworks/base/core/java/android/app/usage/UsageEvents.java - 345.
11. frameworks/base/core/java/android/app/WallpaperInfo.java - 136.

12. frameworks/base/core/java/android/app/WallpaperManager.java - 742, 781, 821.
13. frameworks/base/core/java/android/app/Dialog.java - 372.
14. frameworks/base/core/java/android/app/AliasActivity.java - 83.
15. frameworks/base/core/java/android/app/backup/BackupManager.java - 157.
16. frameworks/base/core/java/android/app/backup/BlobBackupHelper.java - 262.
17. frameworks/base/core/java/android/app/backup/FullBackup.java - 336.
18. frameworks/base/core/java/android/content/AbstractThreadedSyncAdapter.java - 182, 293.
19. frameworks/base/core/java/android/content/AsyncTaskLoader.java - 94, 105.
20. frameworks/base/core/java/android/content/SyncAdaptersCache.java - 86.
21. frameworks/base/core/java/android/content/ContentProvider.java - 242, 265, 280, 326, 341, 357, 573, 589, 400, 422, 443, 459.
22. frameworks/base/core/java/android/content/ContentProviderClient.java - 144, 161, 180, 197, 214, 232, 251, 269, 287, 330, 373, 407, 426, 444.
23. frameworks/base/core/java/android/content/ContentProviderNative.java - 134, 439, 459, 481, 502, 526, 549, 573, 601, 628, 651, 672, 699, 718, 739, 759.
24. frameworks/base/core/java/android/content/ContentProviderOperation.java - 344
25. frameworks/base/core/java/android/content/ContentResolver.java - 342, 397, 535, 584, 620, 1011, 1155, 1240, 1272, 1304, 1336, 1371, 1404.
26. frameworks/base/core/java/android/content/pm/PackageParser.java - 812, 842, 908, 945.
27. frameworks/base/core/java/android/content/pm/ManifestDigest.java - 83.
28. frameworks/base/core/java/android/content/pm/RegisteredServicesCache.java - 129, 573, 637.
29. frameworks/base/core/java/android/content/res/StringBlock.java - 147
30. frameworks/base/core/java/android/database/BulkCursorNative.java - 173, 188, 203, 218, 242, 259, 279.
31. frameworks/base/core/java/android/database/CursorWindow.java - 189, 229, 249, 263, 275, 375, 405, 440, 482, 523, 545, 610, 627, 644, 662, 678, 707.

32. frameworks/base/core/java/android/database/DatabaseUtils.java - 835, 857, 882, 1074, 1141.
33. frameworks/base/core/java/android/database/DefaultDatabaseErrorHandler.java - 91.
34. frameworks/base/core/java/android/database/sqlite/SQLiteConnectionPool.java - 702. frameworks/bas
    - 262, 277.
35. frameworks/base/core/java/android/database/sqlite/SQLiteDatabase.java - 511, 524, 542, 556, 577, 641, 996, 1166, 1319, 1476, 1473, 1501, 1581, 1578, 1681, 1678, 2103, 2108, 2153, 2157
36. frameworks/base/core/java/android/database/sqlite/SQLiteConnection.java - 241, 388, 523, 517, 564, 558, 555, 607, 601, 598, 641, 644, 650, 698 689, 698, 746, 739, 736, 786, 783, 786, 792, 852, 855, 862, 870, 1147.
37. frameworks/base/core/java/android/database/sqlite/SQLiteQuery.java - 73, 76.
38. frameworks/base/core/java/android/database/sqlite/SQLiteSession.java - 350, 442, 590, 621, 654, 687, 723, 757, 791, 840.
39. frameworks/base/core/java/android/database/sqlite/SQLiteStatement.java - 49, 70, 92, 113, 134, 155.
40. frameworks/base/core/java/com/android/internal/app/ToolbarActionBar.java - 453.
41. frameworks/base/core/java/com/android/internal/app/WindowDecorActionBar.java - 1026, 1035.
42. frameworks/base/core/java/com/android/internal/backup/LocalTransport.java - 543.
43. frameworks/base/core/java/com/android/internal/content/PackageHelper.java - 291, 319, 508.
44. frameworks/base/core/java/com/android/internal/net/NetworkStatsFactory.java - 133, 180, 314.
45. frameworks/base/core/java/com/android/internal/os/AtomicFile.java - 174.
46. frameworks/base/core/java/com/android/internal/os/BatteryStatsImpl.java - 8315, 8982.
47. frameworks/base/core/java/com/android/internal/os/ProcessCpuTracker.java - 375, 857.
48. frameworks/base/core/java/com/android/internal/os/RuntimeInit.java - 100.
49. frameworks/base/core/java/com/android/internal/os/SamplingProfilerIntegration.java - 139, 190.
50. frameworks/base/core/java/com/android/internal/os/TransferPipe.java - 94, 119, 161.
51. frameworks/base/core/java/com/android/internal/os/ZygoteConnection.java - 256.

52. frameworks/base/core/java/com/android/internal/os/PowerProfile.java - 251.

53. frameworks/base/core/java/com/android/internal/util/MemInfoReader.java - 33.

54. frameworks/base/core/java/com/android/internal/util/FileRotator.java - 172, 165, 374, 386.

55. frameworks/base/core/java/com/android/server/backup/AccountSyncSettingsBackupHelper.java - 207

9.3 MSC02-J. Generate Strong Random Numbers

Two instances of the java.util.Random class that are created using the same seed will generate identical sequences of numbers. An attacker can learn the value of the seed by performing some observation on the vulnerable target. A more see random number generator, such as the `java.security.SecureRandom` class will fix this problem. The full collection of all instances of non-compliance of the MSC02-J present in binder is listed below:

1. frameworks/base/core/java/android/net/DnsPinger.java - 63

2. frameworks/base/core/java/android/net/NetworkStatsHistory.java - 530

3. frameworks/base/core/java/android/content/ContentResolver.java - 286

9.4 ERR08-J. Do Not Catch NullPointerException or Any of Its Ancestors

A NullPointerException exception thrown at runtime shows the presence of a null pointer dereference that must be fixed in the application code. To fix this problem the solution will explicitly check the String argument for null rather than catching NullPointerException. The full collection of all instances of non-compliance of the ERR05-J present in the binder is listed below:

1. frameworks/base/core/java/android/net/NetworkUtils.java - 379.

2. frameworks/base/core/java/android/net/InterfaceConfiguration.java - 113.

3. frameworks/base/core/java/android/bluetooth/BluetoothDevice.java - 970.

4. frameworks/base/core/java/com/android/internal/net/NetworkStatsFactory.java - 127, 174, 308.

5. frameworks/base/core/java/com/android/internal/os/BatteryStatsImpl.java - 7143.

6. frameworks/base/core/java/com/android/internal/util/XmlUtils.java - 1019, 1038, 1088, 1104, 1148, 1164, 1208, 1224, 1268, 1284, 1473.

9.5 OBJ10-J. Do Not Use Public Static Nonfinal Fields

There is no good reason to declare a field "public" and ""static" without also declaring it "final". Most of the time this is used to share a state among several objects. But with this approach, any object can do whatever it wants with the shared state, such as setting it to null. To overcome this problem use "public static final" instead of "public static". The full collection of all instances of non-compliance of the OBJ10-J present in the binder is listed below:

1. frameworks/base/core/java/android/net/StaticIpConfiguration.java - Line Number 163.
2. frameworks/base/core/java/android/net/netlink/StructNdMsg.java - Line Number 67, 68, 69, 70, 71.
3. frameworks/base/core/java/android/app/UiModeManager.java - 54, 68, 75, 89.
4. frameworks/base/core/java/android/content/pm/ActivityInfo.java - 552.
5. frameworks/base/core/java/com/android/internal/app/ProcessStats.java - 56, 61.
6. frameworks/base/core/java/com/android/internal/app/IntentForwarderActivity.java - 46, 48, 51.
7. frameworks/base/core/java/com/android/internal/widget/Smileys.java - 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61.

9.6 MSC00-J. Use SSLSocket Rather Than Socket for Secure Data Exchange

Programs must use the javax.net.ssl.SSLSocket class rather than the java.net.Socket class when transferring sensitive data over insecure channels. An attacker will get a chance to look at the data if it is plain text. To avoid eavesdropping and malicious tampering the class SSLSocket provides security protocols such as Secure Sockets Layer/Transport Layer Security (SSL/TLS).

1. frameworks/base/core/java/android/net/Network.java - 179.

9.7 FIO02-J. Detect and Handle File-related Errors

Programs that ignore the return values from file operations often fail to detect that those operations have failed. This violation may lead to information disclosure, memory wastage, etc. To overcome this problem consider the return values and handle it in application code itself. The full collection of all instances of non-compliance of the FIO02-J present in the binder is listed below:

1. frameworks/base/core/java/android/app/backup/WallpaperBackupHelper.java - 169
2. frameworks/base/core/java/com/android/internal/backup/LocalTransport.java - 194, 220, 223, 259, 268, 408, 543.
3. frameworks/base/core/java/com/android/internal/util/FileRotator.java - 129, 130, 255, 258, 272, 275, 276, 363
4. frameworks/base/core/java/com/android/server/BootReceiver.java - 267.

10

Appendix B Source Code

In this chapter, the source code used for implementing CERT rules are arranged.

10.1 Read Files Recursively

The tool needs to read thousands of files recursively without human intervention. The following code will read all the java files present in a particular directory and subdirectory.

```
public static void ReadFile(String PrintFname) throws Exception {
 String str;
 FileCount++;
 BufferedReader br = new BufferedReader(new FileReader(PrintFname));
 try {
  StringBuilder sb = new StringBuilder();
  String line = br.readLine();
  while (line != null) {
   sb.append(line);
   sb.append(System.lineSeparator());
   line = br.readLine();
  }
  str = sb.toString();

 } finally {
  br.close();
 }
 if (comFlag == 1) {
  String Cmd = "javac " + PrintFname.toString();
  runProcess(Cmd);
 }
 parse(str, PrintFname);

}
```

10.2 For Generating AST

The CERT rules are implemented based on AST concept. The code listed below will parse the source code and save the AST in variable "cu" of type "CompilationUnit".

```
public static void parse(String str, String PrintFname) throws IOException {
 ASTParser parser = ASTParser.newParser(AST.JLS3);
 parser.setSource(str.toCharArray());
 parser.setKind(ASTParser.K_COMPILATION_UNIT);
 CompilationUnit cu = (CompilationUnit) parser.createAST(null);
 cu.accept(new MyVisitor(cu, str, PrintFname));
}
```

10.3 DCL00-J. Prevent Class Initialization Cycles

The given code saves all the variables used in the constructor definition with the position(Line number). If the initialization is after class object creation it returns DCL00-J violation.

```
 public boolean visit(MethodDeclaration md) {

  if (md.getName().toString().equals(Classname)) {
   md.accept(new ASTVisitor() {
    public boolean visit(Assignment fd) {
     String s = fd.getRightHandSide().toString();
     for (int i = 0; i < VarList.size(); i++) {
      if (s.contains(VarList.get(i))) {
       if (LineList.get(i) > 0 && Clno > 0) {
        if (LineList.get(i) > Clno) {
         PrintError("DCL00-J", LineNo);
         Main.dcl00j++;
        }
       }
      }
     }
     return true;
    }
   });
  }
  return true;
 }
 public boolean visit(FieldDeclaration node) {
   Type t = node.getType();
   if (t.toString().equals(this.Classname) && node.modifiers().toString().contains("static")) {
    Clno = cu.getLineNumber(node.getStartPosition());
   } else if (node.modifiers().toString().contains("static")) {
    node.accept(new ASTVisitor() {
     public boolean visit(VariableDeclarationFragment fd) {
      LineList.add(cu.getLineNumber(fd.getStartPosition()));
      VarList.add(fd.getName().toString());
      return true;
     }
    });
   }
```

10.4 DCL01-J. Do Not Reuse Public Identifiers from the Java Standard Library

Java standard library public identifiers are listed in Oracle website. All these identifiers present in a file named javaclass. A binary search will check if it is used in the program. The given code will do the same.

```
public void ruleDcl01() throws IOException {
 Path filePath = new File("/usr/lib/rasttool/config/javaclass").toPath();
 Charset charset = Charset.defaultCharset();
 List < String > stringList = Files.readAllLines(filePath, charset);
 String[] stringArray = stringList.toArray(new String[] {});
 int retVal = 0;
 String searchVal = this.Classname;
 retVal = Arrays.binarySearch(stringArray, searchVal);
 if (retVal >= 0) {
  PrintError("DCL01-J", LineNo);
  Main.dcl01j++;
 }
}
```

10.5 DCL02-J. Do Not Modify the Collection's Elements During an Enhanced for Statement

The code visits the SingleVariableDeclaration through EnhancedForStatement node and gets the collection variable name to "SingVarDec". If it is present in the Assignment node then it indicates the program is trying to modify the collection's element.

```
public boolean visit(EnhancedForStatement node) {
 if (!node.getParameter().modifiers().toString().contains("final")) {
  dcl02lno = cu.getLineNumber(node.getStartPosition());
  node.accept(new ASTVisitor() {
   public boolean visit(SingleVariableDeclaration SVD) {
    SingVarDec = SVD.getName().toString();
    return false;
   }
  });
  node.accept(new ASTVisitor() {
   public boolean visit(Assignment as) {
    if (as.getLeftHandSide().toString().equals(SingVarDec)) {
     PrintError("DCL02-J", dcl02lno)
     Main.dcl02j++;
    }
    return false;
   }
  });
 }
 return true;
}
```

10.6 EXP02-J. Do Not Use the Object.equals() Method to Compare Two Arrays

ArrList is a list which contains all the arrays present in the program. If the MethodInvocation node is calling "Object.equals()" then the code checks if the parameters present in the ArrList. If yes then triggers EXP02-J violations.

```
 public boolean visit(MethodInvocation node) {
  if (node.toString().contains(".equals") && !node.toString().contains("Arrays.equals")) {
   node.accept(new ASTVisitor() {
    public boolean visit(SimpleName sn) { // Moving inside to get variable
     if (ArrList.contains(sn.getIdentifier().toString())) {
      Flg++;
     }
     return true;
    }
   });
  }
  if (Flg > 1) {
   PrintError("EXP02-J", LineNo);
   Main.exp02j++;
  }
  return true;
 }
```

10.7 ERR02-J. Prevent Exceptions While Logging Data

According to ERR02-J, "console.printf()", "system.out.print", "system.err.print" are dangerous operations which will lead to leakage of sensitive information. These dangerous operations saved in a character array "matches" (like a black list) and checks whether it is present in the catch block MethodInvocation node.

```
  if (node.getException().toString().contains("Throwable")) {
```

```
  catchname = node.getException().getName().toString();
 }
 node.accept(new ASTVisitor() {
  public boolean visit(MethodInvocation mi) {
   String[] matches = new String[] {
    "Console.printf()",
    "System.out.print",
    "System.err.print"
   };
   if (!catchname.equals("")) {
    PrintError("ERR02-J", cu.getLineNumber(mi.getStartPosition()));
    Main.err02j++;
   }
  }
  for (String s: matches) {
   if (mi.toString().contains(s)) {
    mi.accept(new ASTVisitor() {
     public boolean visit(SimpleName sn) {
      if (sn.toString().equals(catchname)) {
       PrintError("ERR02-J", cu.getLineNumber(mi.getStartPosition()));
       Main.err02j++;
      }
      return true;
     }
    });
   }
  }
  return true;
 }
 });
 return true;
 }
```

10.8 ERR04-J. Do Not Complete Abruptly from a Finally Block

ERR04-J says never use return, break, continue, or throw statements within a finally block. The program visits TryStatement and checks any return, break, continue, or throw statement node is present. If there then it returns ERR04-J violation.

```
public boolean visit(TryStatement node) {
  Mil = 0;
  Nil = 0;
  if (node.getFinally() != null) {
   node.getFinally().accept(new ASTVisitor() {
    public boolean visit(BreakStatement sn) {
     PrintError("ERR04-J", cu.getLineNumber(mi.getStartPosition());););
     Main.err04j++;
     return true;
    }
   });
   node.getFinally().accept(new ASTVisitor() {
    public boolean visit(ContinueStatement sn) {
     PrintError("ERR04-J", cu.getLineNumber(mi.getStartPosition());););
     Main.err04j++;
     return true;
    }
   });
   node.getFinally().accept(new ASTVisitor() {
    public boolean visit(ReturnStatement sn) {
     PrintError("ERR04-J", cu.getLineNumber(mi.getStartPosition());););
     Main.err04j++;
     return true;
    }
   });
   node.getFinally().accept(new ASTVisitor() {
    public boolean visit(ThrowStatement sn) {
     PrintError("ERR04-J", cu.getLineNumber(mi.getStartPosition());););
     Main.err04j++;
     return true;
   }
  });
```

10.9 ERR05-J. Do Not Let Checked Exceptions Escape from a Finally Block

The given code visits the finally block in the TryStatement node and checks any MethodInvocation is present without a try-catch. If present if returns ERR05-J violations.

```
public boolean visit(TryStatement node) {
 node.getFinally().accept(new ASTVisitor() {
  public boolean visit(MethodInvocation mi) {

   if (!mi.toString().contains("Log.")) {
    if (!mi.toString().contains("System.")) {
     if (!mi.toString().contains("Console.")) {
      Nil = 1;
      met = mi.toString();
      Mlno = cu.getLineNumber(mi.getStartPosition());
     }
    }
   }
   return true;
  }
 });
 node.getFinally().accept(new ASTVisitor() {
  public boolean visit(TryStatement tr) {
   if (tr.getBody().toString().contains(met)) {
    Mil = 1;
   }
   return true;
  }
 });
 if (Mil == 0 && Nil == 1) {
  PrintError("ERR05-J", cu.getLineNumber(mi.getStartPosition()); Main.err05j++;
  }
 }
 return true;
}
```

10.10 ERR07-J. Do Not Throw RuntimeException, Exception, or Throwable

The given code checks if any class instance creation for RuntimeException, Exception or Throwable is present inside a ThrowStatement node. If there ERR07-J is present.

```
public boolean visit(ThrowStatement node) {
 node.accept(new ASTVisitor() {
  public boolean visit(ClassInstanceCreation cs) {
   cs.accept(new ASTVisitor() {
    public boolean visit(SimpleName sn) {
     if (sn.toString().equals("RuntimeException") || sn.toString().equals("Exception") || sn.toStrin
      PrintError("ERR07-J", LineNo)
      Main.err07j++;
     }
     return true;
    }
   });
   return true;
  }
 });
 return true;
}
```

10.11 ERR08-J. Do Not Catch NullPointerException or Any of Its Ancestors

The program checks the exception present inside a catch clause is NullPointerException. If yes then returns the ERR08-J violation.

```
public boolean visit(CatchClause node) {
 if (node.getException().getType().toString().equals("NullPointerException")) {
  PrintError("ERR08-J", cu.getLineNumber(node.getStartPosition()));
  Main.err08j++;
 }
 return true;
}
```

10.12 FIO02-J. Detect and Handle File-Related Errors

The given code checks if the program is checking the file related errors properly. The SymList2 contains the instances of File class. The ExpressionStatement node with file operation may not check the file related error. So the code flags FIO02-J error.

```
public boolean visit(ExpressionStatement node) {
 if (node.toString().contains(".delete()")) {
  node.accept(new ASTVisitor() {
   public boolean visit(SimpleName as) {
    for (int i = 0; i < SymList1.size(); i++) {
     if (SymList1.contains(as.toString())) {
      if (SymList2.get(i).toString().equals("File")) {
       PrintError("FIO02-J", cu.getLineNumber(node.getStartPosition()));
       Main.fio02j++;
      }
     }
    }
    return false;
   }
  });
 }
 return true;
}
```

10.13 MET09-J. Classes That Define an Equals() Method Must also Define a HashCode() Method

The program checks first save all the method declarations into a list. If an Object.equals is present then it checks a HashCode() is present in the list, if not there MET09-J violation is present.

```
public boolean visit(TypeDeclaration node) {
 Mflag = 0;
 cFlag = 0;
 node.accept(new ASTVisitor() {
  public boolean visit(MethodDeclaration node1) {

   if (node1.getName().toString().equals("equals")) {

    str = node1.toString().replace(node1.getBody().toString(), "");
    if (str.equals("public boolean equals(Object o)")) {
     Mflag = 1;
    }
   }
   return true;
  }
 });
 if (Mflag == 1) {
  node.accept(new ASTVisitor() {
   public boolean visit(MethodDeclaration sn) { // Moving inside to get variable
    String str1 = sn.toString().replace(sn.getBody().toString(), "");
    if (str1.equals("public int hashCode()")) {
     cFlag = 1;
    }
    return true;
   }
  });
 }
 if (cFlag == 0 && Mflag == 1) {
  PrintError("MET09-J", LineNo)
 }
}
}
```

10.14 MSC00-J. Use SSLSocket Rather Than Socket for Secure Data Exchange

The program checks any class instance creation node is present for ServerSocket or Socket class. If it is present then MSC00-J violation is present.

```
public boolean visit(ClassInstanceCreation node) {
 if (node.getType().toString().equals("ServerSocket") || node.getType().toString().equals("Socket"))
  PrintError("NUM07-J", cu.getLineNumber(node.getStartPosition()));
  Main.msc00j++;
 }
 return true;
}
```

10.15 MSC01-J. Do Not Use an Empty Infinite Loop

The program visits the while statement node and checks the expression is true. If true then check the body is empty or not. For empty body MSC01-J violation is present.

```
public boolean visit(WhileStatement stmt) {
 String cntn = stmt.getExpression().toString(); // Get condition True or False
 String sn = stmt.getBody().toString(); // Get body
 sn = sn.replace("{", "");
 sn = sn.replace("}", "");
 sn = sn.replaceAll("\\s+", "");
 if (cntn.equals("true") && sn.equals("")) //Checking condition is true and body is empty
 {
  PrintError("MSC01-J", LineNo);
  Main.msc01j++;

 }

 return true; // do not continue to avoid usage info
}
```

10.16 MSC02-J. Generate Strong Random Numbers

The program checks any class instance creation node is present for Random class. If it is present then MSC02-J violation is present.

```
public boolean visit(ClassInstanceCreation node) {
  if (node.getType().toString().equals("Random")) {
   PrintError("MSC02-J", cu.getLineNumber(node.getStartPosition()));
   Main.msc02j++;
  }
```

10.17 NUM07-J. Do Not Attempt Comparisons with NaN

NaN (not-a-number) is unordered, so the numerical comparison returns false if either or both operands are NaN. The given code checks any NaN operation is present in the code and if yes it returns NUM07-J violation.

```
public boolean visit(SimpleName node) {
  if (node.getIdentifier().toString().equals("NaN")) {
   if (node.getParent() instanceof QualifiedName) {
    QualifiedName ql = ((QualifiedName) node.getParent());
    if (ql.toString().equals("Double.NaN")) {
     PrintError("NUM07-J", cu.getLineNumber(node.getStartPosition()));
     Main.num09j++;
    }
   }
   return true;
  }
```

10.18 NUM09-J. Do Not Use Floating-point Variables as Loop Counters

SymList is a list which contains the data types of all variables. The loop counter variable compares with SymList and if a match is there then NUM09-J violation is present

```
public boolean visit(ClassInstanceCreation node) {
  String s;
  s = node.arguments().toString();
  for (int i = 0; i < SymList1.size(); i++) {
   if (SymList1.contains(s)) {
    if (SymList2.get(i).toString().equals("float")) {
     PrintError("NUM09-J", cu.getLineNumber(node.getStartPosition()));
     Main.num09j++;
    }
   }
   return true;
  }
  public boolean visit(ForStatement node) {
   if (node.initializers().toString().equals("float")) {
    PrintError("NUM09-J", cu.getLineNumber(node.getStartPosition()));
    Main.num09j++;
   }

   return true;

  }
```

10.19 NUM10-J. Do Not Construct BigDecimal Objects from Floating-point Literals

The given code will check any class instance creation for BigDecimal if there then the argument value is a floating point number if will return NUM10-J violation.

```
public boolean visit(ClassInstanceCreation node) {
 if (node.getType().toString().equals("BigDecimal")) {
  if (node.arguments().toString().contains(".") && !node.arguments().toString().contains("\"")) {
   PrintError("NUM10-J", cu.getLineNumber(node.getStartPosition()));
   Main.num10j++;
  }
 }
 return true;
}
```

10.20 OBJ09-J. Compare Classes and Not Class Names

It is insufficient to comparing fully qualified class names because distinct class loaders can load differing classes with identical fully qualified names into a single JVM. The given code will check any method invocation is trying to compare class names, and it returns an error message if the comparison of class names is present.

```
 public boolean visit(MethodInvocation node) {
   if (node.toString().contains(".getClass().getName().equals")) {
    PrintError("OBJ09-J", LineNo);
    Main.obj09j++;
   }
```

10.21 OBJ10-J. Do Not Use Public Static Nonfinal Fields

The rule says do not use public static nonfinal fields. So the program first visits the FieldDeclaration node and get all the modifiers using the method modifiers(). If the “public” “static” “final” is present then corresponding flag values “flag_pub”, “flag_sta”, “flag_fin” will set to 1. If a condition like flag_pub=1, flag_sta=1, flag_fin=0 indicates public static nonfinal field present and it triggers the error message.

```
public boolean visit(FieldDeclaration node) {
 int flag_pub = 0;
 int flag_sta = 0;
 int flag_fin = 0;
 for (int i = 0; i < node.modifiers().size(); i++) {

  if (node.modifiers().get(i).toString().equals("public")) {
   flag_pub = 1;

  }
  if (node.modifiers().get(i).toString().equals("static")) {
   flag_sta = 1;

  }
  if (node.modifiers().get(i).toString().equals("final")) {
   flag_fin = 1;

  }
 }
 if (flag_fin == 0 && flag_pub == 1 && flag_sta == 1) {
  PrintError(OBJ10 - J, Lineno);
  Main.obj10j++;
 }
 return true; //
}
```

10.22 SER05-J. Do Not Serialize Instances of Inner Classes

Programs must not serialize inner classes without a static member classes. The given code checks if any inner class serialization is present without a static keyword. If there it triggers the SER05-J violation.

```
public boolean visit(TypeDeclaration node) {
 Clno = 0;
 SimpleName name = node.getName();
 this.Classname = name.toString();
 this.LineNo = cu.getLineNumber(name.getStartPosition());
 node.accept(new ASTVisitor() {
  public boolean visit(TypeDeclaration nm) {
   if (nm.superInterfaceTypes().toString().equals(node.superInterfaceTypes().toString()) && !nm.supe
    PrintError("SER05-J", LineNo);
   }
   return true;
  }
 });
 return true;

}
```

References

Aho, A. V. and Ullman, J. D. 1977. *Principles of compiler design.* Addison-Wesley.

Android. 2015. The Android source code. `https://source.android.com/source/index.html`.

Association, M. I. S. R. et al. 2008. *MISRA-C: 2004: guidelines for the use of the C language in critical systems.* MISRA.

Black, P. E., Gill, H., Fong, E., et al. 2006. Proceedings of the static analysis summit.

Chess, B. 2007. *Secure programming with static analysis.* Addison-Wesley, Upper Saddle River, NJ.

FbInfer. A tool to detect bugs in Android and iOS apps before they ship. `http://fbinfer.com/`.

iai.uni bonn.de. 2011. Jtransformer query and transformation engine for Java source code. Tech. rep., sewiki.iai.uni-bonn.de. `https://sewiki.iai.uni-bonn.de/research/jtransformer/start`.

Kniesel, G., Hannemann, J., and Rho, T. 2007. A comparison of logic-based infrastructures for concern detection and extraction. In *Proceedings of the 3rd workshop on Linking Aspect Technology and Evolution.* ACM, 6. `https://www.researchgate.net/profile/Guenter_Kniesel/`.

Kuhn, T. 2006. Eye media gmbh, olivier thomann, abstract syntax tree. *Olivier Thomann.*

Long, F., Mohindra, D., Seacord, R. C., Sutherland, D. F., and Svoboda, D. 2011. *The CERT Oracle Secure Coding Standard for Java.* Addison-Wesley Professional.

MITRE. 2012. *MITRE Systems Engineering Guide.* MITRE Corporation.

Olesen, M. C., Hansen, R. R., Lawall, J. L., and Palix, N. 2014. Coccinelle: Tool support for automated cert c secure coding standard certification. *Science of Computer Programming 91*, 141–160.

Oracle. 2014. Secure coding guidelines for Java SE. `http://www.oracle.com/technetwork/java/seccodeguide-139067.html`.

OWASP, T. 2010. 10 2010. *The Ten Most Critical Web Application Security Risks*.

Pech, V., Shatalin, A., and Voelter, M. 2013. Jetbrains mps as a tool for extending java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. ACM, 165–168.

Quinlan, D. 2000. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters 10,* 02n03, 215–226.

Schiela, R. 2014. SEI CERT Coding Standards. `https://www.securecoding.cert.org/confluence/display/seccode/SEI+CERT+Coding+Standards`.

Schmerl, B., Gennari, J., Cámara, J., and Garlan, D. 2016. Raindroid–a system for run-time mitigation of Android intent vulnerabilities.

Seacord, R. C. 2008. *The CERT C secure coding standard*. Pearson Education.

Wikipedia. 2009. Visitor pattern — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Visitor_pattern`.