DEVELOPMENT OF A STATIC ANALYSIS TOOL TO FIND SECURITY VULNERABILITIES IN JAVA APPLICATIONS

A Thesis Submitted to
the Graduate School of Engineering and Sciences of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Engineering

by
Bertan TOPUZ

March 2010
İZMİR

We approve the thesis of **Bertan TOPUZ**

Assist. Prof. Dr. Tuğkan TUĞLULAR
Supervisor

Prof. Dr. Sıtkı AYTAÇ
Committee Member

Prof. Dr. Şaban EREN
Committee Member

12 March 2010

Prof. Dr. Sıtkı AYTAÇ
Head of the Depertment of
Computer Engineering

Assoc. Prof. Dr. Talat YALÇIN
Dean of the Graduate School of
Engineering and Sciences

ACKNOWLEDGEMENTS

I would like to express my profound appreciation and gratitude to my advisor, Assist. Prof. Dr. Tuğkan TUĞLULAR, his valuable guidance, ideas and encouragement in the preparation of this manuscript.

I am very grateful to my committee member, Prof. Dr. Sıtkı AYTAÇ and Prof. Dr. Şaban EREN for their valuable contributions and recommendations.

I would like to thank to my company, Solify-TURKEY for their support.

I would like to express my gratitude to my family for their invaluable patience, understanding, support and encouragements all throughout my life.

ABSTRACT

DEVELOPMENT OF A STATIC ANALYSIS TOOL TO FIND SECURITY VULNERABILITIES IN JAVA APPLICATIONS

The scope of this thesis is to enhance a static analysis tool in order to find security limitations in java applications. This will contribute to the removal of some of the existing limitations related with the lack of java source codes.

The generally used tools for a static analysis are FindBugs, Jlint, PMD, ESC/Java2, Checkstyle. In this study, it is aimed to utilize PMD static analysis tool which already has been developed to find defects Possible bugs (empty try/catch/finally/switch statements), Dead code (unused local variables, parameters and private methods), Suboptimal code (wasteful String/StringBuffer usage), Overcomplicated expressions (unnecessary if statements for loops that could be while loops), Duplicate code (copied/pasted code means copied/pasted bugs). On the other hand, faults possible unexpected exception, length may be less than zero, division by zero, stream not closed on all paths and should be a static inner class cases were not implemented by PMD static analysis tool.

PMD performs syntactic checks and dataflow analysis on program source code. In addition to some detection of clearly erroneous code, many of the “bugs” PMD looks for are stylistic conventions whose violation might be suspicious under some circumstances. For example, having a try statement with an empty catch block might indicate that the caught error is incorrectly discarded. Because PMD includes many detectors for bugs that depend on programming style, PMD includes support for selecting which detectors or groups of detectors should be run. While PMD’s main structure was conserved, boundary overflow vulnerability rules have been implemented to PMD.

ÖZET

JAVA UYGULAMALARINDA GÜVENLİĞE İLİŞKİN ZAYIFLIKLARIN BULUNMASINA YÖNELİK BİR STATİK ANALİZİNİN GELİŞTİRİLMESİ

Bu tez java uygulamalarinda bulunan güvenlik sınırlamaları saptanmasına yönelik statik analiz araçlarını geliştirmeyi amaçlamaktadır. Bu amaçla, java kaynaklı kodlarla ilgili varolan açıkların ortadan kaldırılmasına katkıda bulunacaktır.

FindBugs, Jlint, PMD, Checkstyle ve ESC/Java2 yaygın olarak kullanılan statik analiz araçlarıdır. Bu çalışmada, Possible bugs (olası hatalar), Dead code (kullanılmayan kod), Suboptimal code (yetersiz kod), Overcomplicated expressions (karmaşık ifadeler) ve Duplicate code (ikinci kod) gibi hataların saptanmasına yönelik geliştirilmiş olan PMD statik analiz aracının kullanımı amaçlanmaktadır. Buna karşılık PMD'nin faults possible unexpected exception (olası beklenmeyen kural dışı durum), length may be less than zero(yapı boyutu sıfırdan küçük olabilir), division by zero (sıfıra bölünme), stream not closed on all paths (I/O için açılan katarın kapatılmaması) gibi açıkları bulamadığı saptanmıştır.

Bazı hatalı kodları saptamasının yanında PMD'nin bulmaya çalıştığı hataların çoğu, biçimsel hatalara dayanmaktadır. Örnek olarak, try catch yapısındaki boş catch bloğunu gösterebilmektedir. PMD, programlama stiline bağlı olarak birçok algılayıcı kural içermektedir ve bu kuralları isteğe göre seçebilme özelliğine sahiptir. PMD'nin ana yapısı korunarak taşma zayıflıkları için yeni kurallar geliştirilmiştir.

TABLE OF CONTENTS

APPENDICES

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Static analysis is the automatic method to reason about runtime properties of program code without actually executing it. Properties that will be considered include those which lead to premature termination or ill-defined results of the program, but preclude for instance purely syntactic properties such as syntax errors or simple type errors. Nor does static analysis address errors involving the functional correctness of the software. Hence, static analysis can be used to check that the program execution is not prematurely aborted due to unexpected runtime events, but it does not guarantee that the program computes the correct result. While static analysis can be used to check for e.g. deadlock, timeliness or non-termination there are other, more specialized, techniques for checking such properties; although relying on similar principles.

Static analysis should be contrasted with dynamic analysis which concerns analysis of programs based on their execution, and includes e.g. testing, performance monitoring, fault isolation and debugging.

Static analysis is useful in several respects. It can be used to detect certain types of software runtime errors e.g. division by zero, arithmetic overflows, array indices out of bounds, buffer overflows etc without actually executing the code. However, static analysis does not in general guarantee the absence of runtime errors. While static analysis can reduce the need for testing or even detect errors that in practice cannot be found by testing, it is not meant to replace testing.

In addition to finding errors, static analysis can also be used to produce more efficient code; in particular for safe languages like Java, where efficiency was not the primary goal of the language designers. Many runtime tests carried out in Java programs can in practice be avoided given certain information about the runtime behavior. For instance, tests that array indices are not out-of-bounds can be omitted if it is known that the value of the indices is limited to values in-bounds. Static analysis can provide such information.

Static analysis can also be used for type inference in untyped or weakly typed languages or type checking in languages with non-static type systems. Finally static analysis can be used for debugging purposes, automatic test case generation, impact analysis, and intrusion detection and for software metrics.

In this thesis, we will focus on to find security vulnerabilities in java applications.

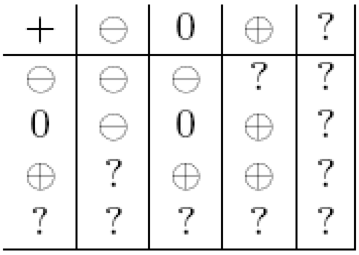
CHAPTER 2

STATIC ANALYSIS

2.1. Principles of Static Analysis

Some properties checked by static analysis tools can be carried out by relatively straightforward pattern matching techniques. However, most properties are more challenging and require much more sophisticated analysis techniques. It is often claimed that static analysis is done without executing the program, but for nontrivial properties this is only partially true. Static analysis usually implies executing the program not in a standard way, but on an abstract machine and with a set of abstract non-standard values replacing the standard ones. The underlying concept is that of a state; a state is a collection of program variables and the association of values to those variables. State information is crucial to determine if a statement such as x=x/y may result in division by zero (it may do so if y may have the value zero at the time when the division is made). In the case of an intra-procedural analysis the state takes account only of local variables while a context-sensitive analysis must take account also of global variables plus the contents of the stack and the heap. The program statements are state transformers and the aim of static analysis is to associate the set of all possible states with all program points. Such sets of states are typically infinite or at least very large and the analysis must therefore resort to some simplified description of the sets representing only some of the relationships between the program variables, e.g. tracking an interval from which a variable may take its value.

For instance, instead of computing with the integers it may be computed with values that describe some property of the integers; may as a simple example replace the domain of integers with the finite domain {Θ, 0,⊕ ,?} where Θ designates a negative integer (i.e. the interval]-∞ ,-1]), 0 designates the integer 0 (the interval[0, 0]), ⊕ designates a positive integer (the interval [1,∞ [) and ? designates any integer (the interval]- ∞, ∞ [). Operations, such as addition, which normally operate on the integers, must be redefined over the new domain and in such a way that the abstract operation mimics the concrete one in a faithful way. The abstract operation of addition can be defined as in Figure 2.1 (Emanuelsson and Nilsson, 2008).



+	⊖	0	⊕	?
⊖	⊖	⊖	?	?
0	⊖	0	⊕	?
⊕	?	⊕	⊕	?
?	?	?	?	?

Figure 2.1. The abstract operation of addition
(Source: Emanuelsson and Nilsson, 2008)

Such abstractions leads to loss of information which influences the precision of the analysis; if it is known that x = 4 and y = -3 then x + y is positive, but if it is only known that x = ⊕ and y = Θ then it is inferred that x + y is an integer.

2.2. Types of Static Analysis

2.2.1. Secure Information Flow

Conventional security mechanisms such as access control and encryption do not directly address the enforcement of information-flow policies. Recently, a promising new approach has been developed: the use of programming-language techniques for specifying and enforcing information-flow policies (Sabelfelt and Myers, 2003). Security mechanisms of most computer systems make no attempt to guarantee secure information flow. "Secure information flow," or simply "security," means here that no unauthorized flow of information is possible. In the common example of a government or military system, security requires that processes be unable to transfer data from files of higher security classifications to files (or users) of lower ones: not only must a user be prevented from directly reading a file whose security classification exceeds his own, but he must be inhibited from indirectly accessing such information by collaborating in arbitrarily ingenious ways with other users who have authority to access the information Most access control mechanisms are designed to control immediate access to objects without taking into account information flow paths implied by given, outstanding collection of access rights. Contemporary access control mechanisms, have demonstrated their abilities to enforce the isolation of processes essential to the success of a multitask system. These systems rely primarily on assumptions of "trustworthiness" of processes for secure information flow among cooperating processes (Denning, 1976).

The most important objectives of information security systems are to protect confidentiality, integrity, and availability. It is obvious that compromises in integrity are the main causes of compromises in confidentiality and availability. Therefore, mechanism that specifies and enforces secure information flow policies is needed within application programs (Huang et al., 2004).

Information flow is a way of modelling data flow within a program usually meaning that values can flow in one direction (in security this is from safe to unsafe). This is usually looked at regarding security policies (England, 2008).

Type systems have proven useful for specifying and checking program safety properties and also used to verify program security. By means of programmer-supplied annotations, both proof-carrying codes (PCC) (Necula, 1997) and typed assembly languages (TAL) (Morrisett et al., 1999) are designed to provide safety proofs for low-level compiler-generated programs.

Many security verification efforts have focused on temporal safety properties related to control flow.

2.2.1.1. Type-Based Analysis

Since vulnerabilities in Web applications are primarily associated with insecure information flow, the use of proper information flow rather than control flow is generally accepted. The first widely accepted model for secure information flow was given by Bell and La Padula (McLeen, 1990). They stated two axioms: a) a subject cannot access information classified above its clearance, and b) a subject cannot write to objects classified below its clearance. Their original model only dealt with confidentiality. Denning (1976) established a lattice model for analyzing secure information flow in imperative programming languages based on a program abstraction derived from an *instrumented semantics* of a language. Orbaek (1995) proposed a similar treatment, but addressed the secure information flow problem in terms of data integrity instead of confidentiality. To base directly on standard language semantics, Volpano et al. showed that Denning's axioms can be enforced using a type system in which program variables are associated with security classes that allow inter-variable information flow to be statically checked for correctness. Soundness was proven by showing that well-typed programs ensure confidentiality in terms of *non-interference*. Recently, fully functional type systems designed to ensure secure information flow have been offered for high-level, strong-typed languages such as ML (Pottier and Simonet, 2003) and Java (Myers, 1999).

Type-based approaches to static program analysis are attractive because they prove program correctness without unreasonable computation efforts. Their main drawback is their high false positive rate, which often makes them impractical for real-world use. Regardless of whether security classes are assigned through manual annotations or through inference rules, they are statically bound to program variables in conventional type systems. It is important to keep in mind that the security class of a variable is a property of its state, and therefore varies at different points or call sites in a program. In JIF and similar type-based systems, variable labels become increasingly restrictive during computation, resulting in high false positive rates. JIF addresses this problem by giving programmers the power to *declassify* variables—that is, to explicitly relax variable labels.

Type analysis started with the syntax tree of a program and defined constraints over variables assigned to nodes. Analyses that work in this manner are flow insensitive, in the sense that the results remain the same if a statement sequence S_1S_2 is permuted into S_2S_1. Analyses that are flow sensitive use a control flow graph, which is a different representation of the program source. A control flow graph (CFG) is a directed graph, in which nodes correspond to program points and edges represent possible flow of control. A CFG always has a single point of entry, denoted entry, and a single point of exit, denoted exit. If v is a node in a CFG then *pred(v)* denotes the set of predecessor nodes and *succ(v)* the set of successor nodes. The CFGs for assignments, output, return statements, and declarations are given in Figure 2.2.

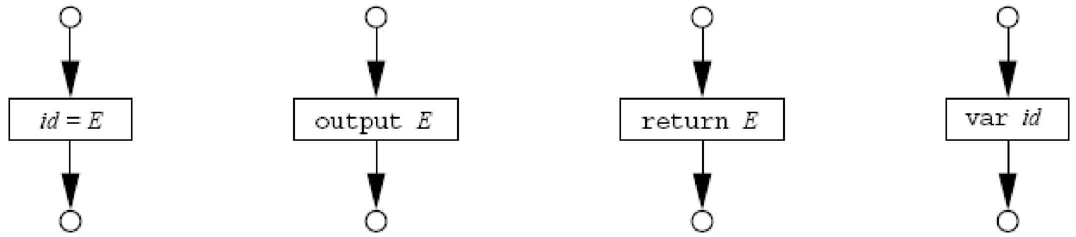


Figure 2.2. CFGs for assignments, output, return statements, and declarations.

For the sequence S_1 S_2, the exit node of S_1 and the entry node of S_2 are eliminated and combined the statements together as shown in Figure 2.3.

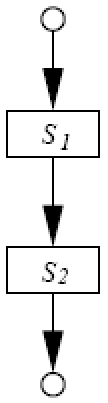


Figure 2.3. Sequence of S_1 S_2.

Similarly, the other control structures are modelled by inductive graph constructions as given in Figure 2.4.

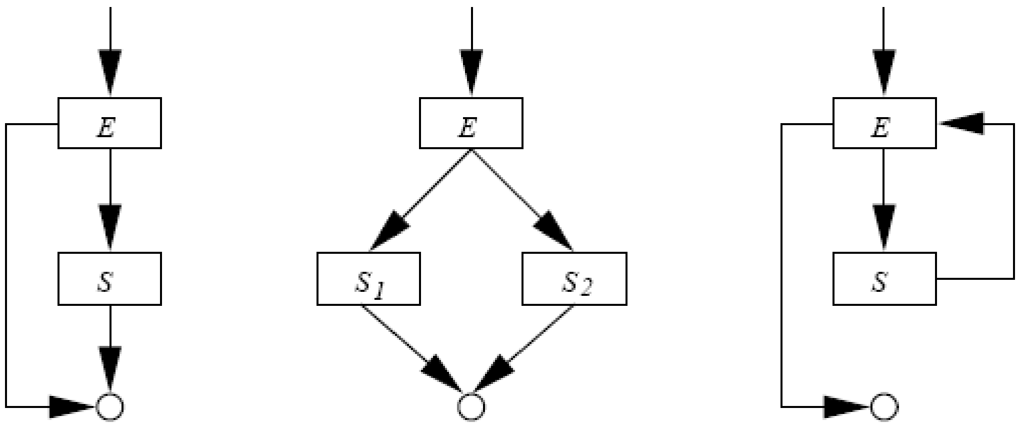


Figure 2.4. Inductive graph constructions.

Using this systematic approach, CFG of the iterative factorial function is given in Figure 2.5.

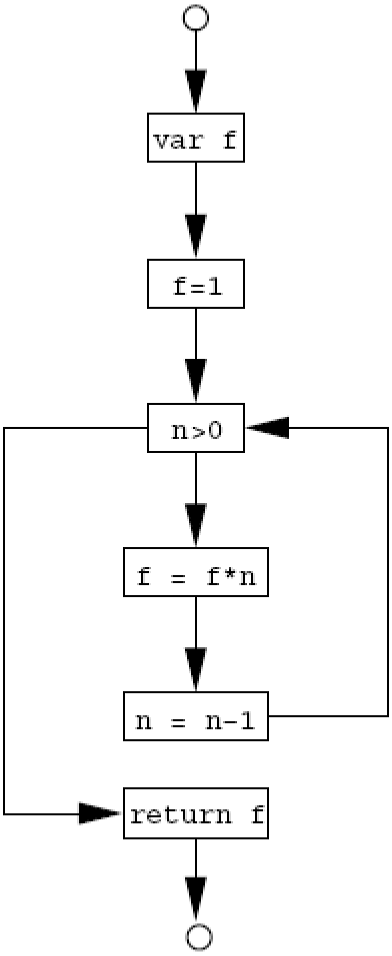


Figure 2.5. CFG of the iterative factorial function.

2.2.2. Data Flow Analysis

The purpose of data flow analysis is to statically compute certain information for every single program point (or for coarser units such as functions). For instance, the classical *constant analysis* computes, for each program point, the literal values that variables may hold. Classical dataflow analysis, also called the monotone framework, starts with a Control Flow Graph (CFG) and a lattice L with finite height. The lattice may be fixed for all programs, or it may be parameterized with the given program. To every node v in the CFG, it can be assigned a variable [v] ranging over the elements of L. For each construction in the programming language, a dataflow constraint that relates the value of the variable of the corresponding node to those of other nodes (typically the neighbours) is defined. As for type inference, it will be ambiguously used the notation [S] for [v] if S is the syntax associated with v.

For a complete CFG, a collection of constraints can be systematically extracted over the variables. If all the constraints happen to be equations or inequations with monotone right-hand sides, then it can be used the fixed-point algorithm to compute the unique least solution. The dataflow constraints are sound if all solutions correspond to correct information about the program. The analysis is conservative since the solutions may be more or less imprecise, but computing the least solution will give the highest degree of precision.

2.2.2.1 Fixed Point Algorithm

If the CFG has nodes $V = \{v_1, v_2, . . . , v_n\}$, with lattice L_n. Assuming that node v_i generates the dataflow equation $[v_i] = F_i([v_1], . . . , [v_n])$, and combined function F : $L^n \rightarrow L^n$ described as:

$F(x_1, . . . , x_n) = (F_1(x1, . . . , x_n), . . . , F_n(x_1, . . . , x_n))$

The naive algorithm is then to proceed as follows:

```
$x = (\perp, . . . , \perp)$;
do { t = x; x = F(x); }
while ($x \neq t$);
```

to compute the fixed-point x. A better algorithm, called chaotic iteration, exploits the fact that lattice has the structure L^n:

```
$x1 = \perp$; . . . $x_n = \perp$;
do {
$t_1 = x_1$; . . . $t_n = x_n$;
$x_1 = F_1(x_1, . . . , x_n)$;
. . .
$x_n = F_n(x_1, . . . , x_n)$;
} while ($x_1 \neq t_1$ V . . . V $x_n \neq t_n$);
```

to compute the fixed-point $(x_1, . . . , x_n)$.

2.2.2.2. Forwards-Backwards-May-Must

Four classical analyses can be compared in various ways. They are all just instances of general monotone framework, but their constraints have a particular structure. A forwards analysis is one that for each program point computes information about the past behaviour. Examples of this are available expressions and reaching definitions. They can be characterized by the right-hand sides of constraints only depending on predecessors of the CFG node. Thus, the analysis starts at the entry node and moves forwards in the CFG.

A backwards analysis is one that for each program point computes information about the future behaviour. Liveness and very busy expressions are the examples. They can be characterized by the right-hand sides of constraints only depending on successors of the CFG node. Thus, the analysis starts at the exit node and moves backwards in the CFG.

A may analysis is one that describes information that may possibly be true and, thus, computes an upper approximation. Examples of this are liveness and reaching definitions. They can be characterized by the right-hand sides of constraints using a union operator to combine information.

A must analysis is one that describes information that must definitely be true and, thus, computes a lower approximation. Examples of this are available expressions and very busy expressions. They can be characterized by the right-hand sides of constraints using an intersection operator to combine information. All possible combination is given in Table 2.1.

Table 2.1. The classic data flow analyses.

	Forward	Backward
May	*Reaching definitions* The assignments that produced current variable values	*Live variables* Variables whose current values may be used later
Must	*Available expressions* Computed expressions whose values have not Changed	*Very busy expressions* Expressions that are always evaluated (in a loop)

2.2.3. Mutability Analysis

A mutability analysis determines which fields and objects are mutable, and which methods may mutate them. Knowing which method parameters may be mutated during a method's execution is useful for many software engineering tasks such as modelling, verification, compiler optimizations, and program transformations such as refactoring, test input generation, and regression oracle creation as well as specification mining. Both static and dynamic analysis techniques have been employed to detect immutable parameters (Artzi et al., 2007). Computing accurate static analysis approximations threatens scalability, and imprecise approximations can lead to weak results. Dynamic analyses offer an attractive complement to static approaches, both in not using approximations and in detecting *mutable* parameters.

The goal of parameter mutability analysis is the classification of each method parameter (including the receiver) as either reference mutable or reference-immutable. Informally, reference immutability guarantees that a given reference is not used to modify its referent. Parameter *p* of method *m* is reference-*mutable* if there exists an execution of *m* in which *p* is *used* to mutate the state of the object pointed to by *p*. Parameter *p* is said to be *used* in a mutation, if the left hand side of the mutating assignment was obtained during the given execution via a series of field accesses and copy operations from *p*. If no such execution exists, the parameter *p* is reference-*immutable*.

Static mutability analysis consists of two phases: S, an intraprocedural analysis that classifies as (*im*)*mutable* parameters (never) affected by field writes within the procedure itself, and P, an interprocedural analysis that propagates mutability information between method parameters. P may be executed at any point in an analysis pipeline after S has been run, and may be run multiple times (interleaving with other analyses). S and P both rely on an intraprocedural pointer analysis that calculates the parameters pointed to by each local variable.

An intraprocedural, context-insensitive, flow-insensitive, 1-level field-sensitive, and points-to analysis are used to determine which parameters can be pointed to by each expression. As a special case, the analysis is flow-sensitive on the code from the beginning of a method through the first backwards jump target, which includes the entire body of methods without loops. The points-to analysis calculates, for each local variable l, a set $P_0(l)$ of parameters whose state l can point to directly and a set $P(l)$ of parameters whose state l can point to directly or transitively. The points-to analysis has "overestimate" and "underestimate" varieties; they differ in how method calls are treated.

The static analysis S works in four steps.

(1) S performs the "overestimate" points-to analysis.

(2) The analysis marks as *mutable* some parameters that are currently marked as *unknown*: For each mutation $l_1. f = l_2$, the analysis marks all elements of $P_0(l_1)$ as *mutable*.

(3) The analysis computes a "leaked set" L of locals, consisting of all arguments (including receivers) in all method invocations and any local assigned to a static field (in a statement of the form Global.field = local).

(4) The analysis marks as *immutable* all *unknown* parameters that are not in the set $\cup_{l\in L}$ *P(l)* only if all method's parameters can be marked *immutable*.

S is i-sound and m-unsound. To avoid over-conservatism, S assumes that on the entry to the analyzed method all parameters are fully un-aliased, i.e., point to disjoint parts of the heap. This assumption may cause S to miss possible mutations due to aliased parameters; to maintain i-soundness, S never classifies a parameter as *immutable* unless all other parameters to the method can be classified as *immutable*. The m-unsoundness of S is due to infeasible paths (e.g., unreachable code), flow-insensitivity, and the overestimation of the points-to analysis.

The interprocedural propagation phase P refines the current parameter classification by propagating both mutability and immutability information through the call graph. Given an i-sound input classification, propagation is i-sound and m-unsound.

Because propagation ignores the bodies of methods, the P phase is i-sound only if the method bodies have already been analyzed.

2.2.4. Points to Analysis

The most important information that must be obtained is the set of possible targets of pointers. There are infinitely many possible targets during execution, so it must be selected some finite representatives. The canonical choice is to introduce a target &id for every variable named id and a target malloc-i, where i is a unique index, for each different allocation site (program point that performs a malloc operation). Targets are used to denote the set of pointer targets for a given program.

Points-to analysis takes place on the syntax tree, since it will happen before or simultaneously with the control flow analysis. The end result of points-to analysis is a function pt that for each (pointer) variable p returns the set pt (p) of possible pointer targets to which it may evaluate. A conservative analysis could be performed, so these sets will in general be too large. Given this information, many other facts can be approximated. If it is determined whether pointer variables p and q may be aliases, then a safe answer is obtained by checking whether pt(p)∩pt(q) is non-empty. The simplest analysis possible, called address taken, is to use all possible targets, except that &id is only included if this construction occurs in the given program. This only works for very simple applications, so more ambitious approaches are usually preferred. If there is any restriction in typable programs, then any points-to analysis could be improved by removing those targets whose types are not equal to that of the pointer variable.

2.2.5. Dependence Analysis

Fundamental analysis step in an advanced optimizing compiler (as well as many other software tools) is data dependence analysis for arrays. This means deciding if two references to an array can refer to the same element and if so, under what conditions. This information is used to determine allowable program transformations and optimizations.

Dependence analysis is a very important part of any vectorizing or concurrentizing compiler. Data dependence analysis has great importance for the automatic detection and exploitation of implicit parallelism. Therefore, experimental evaluation to determine the accuracy of dependence analysis techniques is very important. Such evaluation is necessary to guide research and to help compiler writers in the selection of a dependence analysis strategy. Constant test, the GCD (Greatest Common Divisor) test, three variants of Banerjee's inequalities, and integer-programming based tests such as the Omega test are the techniques that included in dependence analysis.

Dependence analyses generally focus on statements with array references and assume that the two statements to be analyzed are both inside the same, possibly multiply-nested, DO loop. The generic loop test can be summarized as (Petersen and Padua, 1993);

```
 DO $I_1=L_1,U_1$
          . . .
          DO $I_d = L_d,U_d$

$S_v$;  $X(f_1 (I_l,...,I_d),……… f_n (I_l,...,I_d)) =$
$S_w$;      $…= X(g_1 (I_l,...,I_d),……… g_n (I_l,...,I_d)) =$


          END DO
          . . .
     END DO
```

Where X is an n-dimensional array and f_i and g_i are the functions from Z^d to Z.

The constant test is the only approximate method that not only can break dependence but can also conclusively prove dependence. If all the subscripts in the two array references are loop invariant and have the same value, then there will be data dependence for all potential direction vectors. If any pair of corresponding subscripts is constant and different, then there is no data dependence regardless of the values of any other subscript. Loop invariant expressions that are common to both subscripts in the potential dependence are cancelled before the comparison is made.

The greatest common divisor (GCD) test establishes an existence criterion for the solution to the equation F_i(I’) = g_i(I”). This test is based on the fact that when both f_i and g_i are linear, a solution to the equation exists if the greatest common divisor of the coefficients of I’ and I“ also divides the constant term. Conversely, if it does not divide the constant term, then no solution can exist. When the GCD method breaks a potential dependence, it breaks all the direction vectors simultaneously. The GCD method cannot prove dependence because it does not take into account the value of the loop limits.

The Omega test combines new methods for eliminating equality constraints with an extension of Fourier-Motzkin variable elimination to integer programming. The Omega test determines whether there is an integer solution to an arbitrary set of linear equalities and inequalities, referred to as a problem. In addition to supporting the full capabilities of integer-programming, the Omega test also permits the systematic handling of unknown additive terms. Consider the subscripts X(l+N) and X(I') where $1 \leq I$, $I' \leq N$ (N is the loop upper limit). The Omega test is capable of analyzing such expressions involving unknown additive constrained variables. After the addition of the loop limit, it can be found that the system of equations is inconsistent since $I + N \neq I'$ for all $I^{'}$ in [1... N]. Pugh (1992) found that the time required by the Omega test to analyze a problem is rarely more than twice the time required to scan the array subscripts and loop bounds. This would indicate that the Omega test is suitable for use in production compilers.

2.2.6. Escape Analysis

Escape analysis is a static analysis that determines whether the lifetime of data may exceed its static scope. The main originality of escape analysis is that it determines precisely the effect of assignments, which is necessary to apply it to object oriented languages with promising results.

Object-oriented languages such as C++ and Java often use a garbage collector (GC) to make memory management easier for the programmer. A GC is even necessary for the Java programming language, since Java has been designed to be safe, so it cannot rely on the programmer to deallocate objects when they are useless. However, garbage collecting data is time consuming, especially with a mark and sweep collector as in the JDK (Java Development Kid). Therefore stack allocation may be an interesting alternative. However, it is only possible to stack allocate data if its lifetime does not exceed its static scope. The goal of escape analysis is precisely to determine which objects can be stack allocated.

Escape analysis is an abstract interpretation-based analysis which has been already applied to functional languages. However, object-oriented languages have specific features, which make the analysis completely different from the functional version:

- Object-oriented languages use dynamic calls, so before analyzing the code, it must be first determined which methods may actually be called at each call point;
- Object-oriented languages make an intensive use of assignments, which must therefore be precisely analyzed, which much complicates our task;
- Object-oriented languages use sub-typing, which must be taken into account for the representation of escape information, since it is computed from the types.

Escape analysis has two applications: an object *o* that does not escape from method *m* (whose lifetime is limited to the execution of *m*) can be stack allocated in *m*. Moreover, when analysis claims that *o* does not escape from *m*, *o* is also local to the current thread, so it did not needed do synchronizations when calling a synchronized method on object *o*. This second optimization is important, because synchronization is a costly operation in the JDK. Moreover, there is much synchronization even in single-threaded programs since libraries use synchronization to ensure thread-safety in all cases. Synchronization elimination could also be applied to other multithreaded languages.

2.2.7. Alias Control and Confinement

Confinement properties impose a structure on object graphs which can be used to enforce encapsulation properties. From a practical point of view, encapsulation is essential for building secure object-oriented systems as security requires that the interface between trusted and untrusted components of a system be clearly delineated and restricted to the smallest possible set of operations and data structures. One of the main benefits of object-oriented programming is information hiding and encapsulation: classes and visibility controls offer encapsulation without sacrificing extensibility and flexibility. But it is well known that the unfettered use of shared references to mutable objects is error-prone and can violate intended encapsulation. Quite a few proposals have been made for confinement (alias control).

One useful confinement property is unique (unshared, references). These have strong properties but for many purposes sharing is needed. Sharing can often be confined to the scope of a module (a sealed or closed package). This facilitates a coarse form of encapsulation that is quite useful, e.g., in debugging and security. Stronger and more fine-grained confinement is needed for reasoning about specifications using “modified clauses” and for justifying program transformations.

Banerji et al., defined the confinement as a property of program syntax that is a static analysis of instance-based confinement. The definition was syntax-directed and did not require any code annotations or flow analysis.

The syntax is based on that of Java, with some restrictions for ease of formalization; for example, “return” statements appear only at the end of a method, and heap effects (new and field update) occur in commands rather than expressions. There are a couple of minor deviations from Java, e.g., the keyword var marks local variable declarations. A program consists of a collection of class declarations like the following one.

```
class Boolean extends Object      {
bool f;
unit set(bool x){ this.f := x; return unit }
bool get()  { skip; return this.f } }
```

Instances of class Boolean has a private field f with the primitive type bool. There is no constructor; fields of new objects are given their Java defaults (null, false). Fields are considered to be private to their class and methods public: fields are only visible to methods declared in this class, but methods are visible to all classes. Fields are accessed in expressions of the form this.f, using “this” to refer to the current object. The unit value; the singleton type, unit, corresponds to Java's “void". Object types are implicitly references: assignment creates aliases and == compares references.

The primary challenge in building and evolving large object-oriented systems is understanding aliasing between objects. Unexpected aliasing can lead to broken invariants, mistaken assumptions, security holes, and surprising side effects, all of which may lead to software defects and complicate software evolution (Aldrich et al., 2002). Aldrich defined three major challenges in order to bring alias control into practice (Clark et al., 2008);

1. The community has to identify applications where the benefit of making program structure explicit has a significant and immediate benefit. Two promising candidates are concurrency and verification. For both applications it will be necessary to improve the expressiveness in order to common program styles and idioms.
2. The community has to increase the adoptability of alias control by reducing the annotation burden through inference and by providing support for existing languages and programs.
3. The community has to increase the applicability of alias control to be able to handle more programs. He mainly stated that the community has focused too much on restricting aliasing rather that documenting the aliasing programs and using information for reasoning.

Understanding and evolving large software systems is one of the most pressing challenges confronting software engineers today. When evolving a complex system in the face of changing requirements, developers need to understand how the system is organized in order to work effectively. For example, to avoid introducing program defects, programmers need to be able to predict the effect of making a software change. Also, while fixing defects, programmers need to be able to track value flow within a program in order to understand how an erroneous value was produced. In an object-oriented program, all of these tasks require understanding the data sharing relationships within the program. These relationships may be very complex at worst, a reference could point to any object of compatible type and current languages do not provide much help in understanding them.

Data sharing problems can also compromise the security of a system. For example, in version 1.1 of the Java standard library, the security system function Class.getSigners() returned a pointer to an internal array, rather than a copy. Clients could then modify the array, compromising the security of the "sandbox" that isolates Java applets and potentially allowing malicious applets to pose as trusted code. Existing languages provide poor support for preventing security problems that arise from improper data sharing. Aldrich et al., described the AliasJava annotation system to understand the data sharing patterns in Java programs. The annotations bound aliasing on the heap structurally: *unique* describes an unshared reference, *owned* objects are assigned an *owner* that controls who may access that object, and *shared* indicates the worst case of a globally-aliased reference. It is also provided a *lent* annotation expressing sharing that is temporally bounded by the length of a method call.

2.3. Static Analysis Tools

Static analyzers are used to discover difficult to find programming errors before run time when they may be more difficult or impossible to find. This class of tool can discover many logical and security errors in an application without executing the compiled application. Unlike dynamic analysis tools which look at the application state while it is being executed, static analysis tools do not require the application to be compiled or executed; bugs can be found by analyzing the source code directly.

Moreover, static techniques can explore abstractions of all possible program behaviours, and thus are not limited by the quality of test cases in order to be effective. Using static analysis to find bugs has some advantages over the traditional quality assurance techniques of testing and manual code inspections. Unlike testing, static analysis can easily check hard-to-execute code paths such as error-handling code. Compared to manual code inspection, static analysis is less easily confused by what code appears to do, and is relatively inexpensive to apply. For these reasons, static analysis to find bugs is a very active research area, and increasingly is becoming a standard part of the quality assurance toolbox in development of projects.

Static techniques range in their complexity and their ability to identify or eliminate bugs. The most effective (and complex) static technique for eliminating bugs is a *formal proof* of correctness. While the existence of a correctness proof is perhaps the best guarantee that the program does not contain bugs, the difficulty in constructing such a proof is prohibitive for most programs. *Partial verification* techniques have been proposed. These techniques prove that some desired property of a program holds for all possible executions. Such techniques may be complete or incomplete; if incomplete, then the analysis may be unable to prove that the desired property holds for some correct programs. Finally, *unsound* techniques can identify “probable” bugs, but may miss some real bugs and also may emit some inaccurate warnings (Hovemeyer and Pugh, 2004).

Static analysis tools have been used in a rudimentary form for the majority of the history of modern programming languages. Early versions of simple pattern matching static analysis tools have been used to enforce coding styles within a company, or to discover simple programming errors. As more research in the subject was completed, developers of static analysis tools found more programmatic errors that could be discovered. Today tools can scan C, C++ and Java code for many common coding problems within many different categories (Ware and Fox, 2008).

Static analysis tools have gone through many stages of sophistication. In their infancy the tools were little more than a pattern matching command such as grep. The programmer could search for a list of functions that were known to be dangerous and which should be avoided. In this early stage the tools were difficult to use, tedious, and limited in their ability to find real bugs (Hovemeyer and Pugh, 2004).

The next attempt at finding bugs using static analysis techniques came by looking at code metrics, such as lines of code, ratio of lines of code to lines of comments, cyclomatic complexity and others. Using these techniques the developer could gain a greater understanding of the complexity of the code. Complexity metrics such as lines of code per function could help a developer break the code into smaller parts for greater readability or lesser complexity. Cyclomatic complexity is one of the more widely used software quality metrics. It allows a developer or tester to measure the potential for bugs in a program, by mapping the number of independent paths through each module. The more paths that can be taken, the more complex the code is, and the more likely that there will be bugs waiting to be found (McCabe, 1976; Shepperd, 1988).

The next step in the evolution of static code analysis was to use more sophisticated searching algorithms. By adding some context to the search it became possible to find bugs that required interaction between multiple function calls, such as usage of alloc without a matching free, failing to close an open network connection and many others. Tables could be employed to ensure each memory allocation was being properly de-allocated and that it was the same memory reference allocated in the beginning.

Static analysis tools then began adding Semantic Analysis techniques (Barbuti et al., 1993) that enabled discovery of the basic structure and relation of each function within the application. This additional contextual information helps the analyzer understand and report bugs that require knowledge of specific code paths through the application. The most advanced static analyzers use abstract syntax trees to provide the best possible bug finding capabilities. Crew (1997) describes a language for specifying patterns to match the abstract syntax trees of C and C++ programs (ASTLOG). ASTLOG has been used successfully to find bugs and performance problems, and is the basis of the PREfast tool used extensively within Microsoft to find bugs. ASTLOG is an interesting data point in the space of static analysis techniques to find bugs, because it shows that simple pattern-matching approaches can be very effective at finding interesting program features, such as probable bugs. Unlike dataflow analysis and abstract interpretation, this kind of analysis does not directly model the semantics of program operations, focusing entirely on syntactic structures. Even given this limitation, ASTLOG has been used successfully to find bugs. Using the knowledge gained by building an abstract syntax tree, a static analysis tool can run detailed simulations of suspicious code fragments to better predict how the code will react at runtime.

Some static analysis tools allow developers to mark their code with special comments or some other form of metadata to describe rules and inter-function dependencies. This additional information allows the analyzer to understand under what conditions a bug may occur as well as expectations each function has for parameters passed in and values returned. The use of metadata keeps the number of false positives down and helps the analyzer follow code paths more closely.

Recently static analysis tools have allowed developers or testers to create their own rules or modify existing rules or plug-ins. These customizations can help tailor the static analyzer specifically to the target application. This customization enables a developer to look for bugs specific to their operating environment, application needs, and coding standards.

2.3.1. How Static Analysis Tools Find Flaws

The first thing a static analysis tool must do is identify the code to be analyzed. The source files that must be compiled to create a program may be scattered across many directories, and may be mixed in with other source code that is not used for that program. Static analysis tools operate much like compilers so they must be able to identify exactly which source files contribute and should ignore those that do not. The scripts or build system that builds the executable obviously know which files to use, so the best static analysis tools can extract this information by reading those scripts directly or by observing the build system in action. This way the tool gets to see not only the source files but also which compiler is being used and any command-line flags that were passed in. The parser that the static analysis tool uses must interpret the source code in the same way that the real compiler does. It does this by modeling how the real compile works as closely as possible. The command-line flags are an essential input to that.

As the build system progresses, each invocation of the compiler is used to create a whole program model of the program. This model consists of a set of abstract representations of the source, and is similar to what a compiler might generate as an intermediate representation. It includes the control-flow graph, the call graph, and information about symbols such as variables and type names.

Once the model has been created, the analysis performs a symbolic execution on it. This can be thought of as a simulation of a real execution. Whereas a real execution would use concrete values in variables, the symbolic execution uses abstract values instead. This execution explores paths and, as it proceeds, if any anomalies are observed, they are reported as warnings. This approach is based on abstract interpretation (Cousot and Cousot, 1977) and model checking (Cousot and Cousot 1999; Clarke et al., 2002). Abstract interpretation, introduced was developed to provide a firmer theoretical foundation for dataflow analysis. An abstraction function maps the infinite domain of program objects into a finite domain of abstract objects. The effects of program operations are modeled using the abstract domain. A concretization function maps the abstract objects back into the program domain (with some loss of precision). Dataflow analysis can be considered a special case of abstract interpretation. Model checking is a technique for exploring reachable states in a concurrent system for errors. It was initially developed for verification of hardware systems; recent research has applied it to software systems. The SLAM project (Ball and Rajamani, 2002) applies model checking to boolean abstractions of programs. For predicates in a C program, boolean predicates are abstracted such that if a property is true of the Boolean program, it will also be true of the original C program. The boolean abstraction of the original program is done incrementally; if the desired property cannot be proved, new predicates are added and the boolean program checked again.

The analysis is *path-sensitive*, which means that it can compute properties of individual paths through the program. This is important because it means that when a warning is reported, the tool can tell the user the path along which execution must proceed in order for the flaw to be manifest. Tools also usually indicate the points along that path where relevant transformations occur and conditions on the data values that must hold. These help users understand the result and how to correct the problem should it be confirmed.

Once a set of warnings have been issued, these tools offer features to help the user manage the results, including allowing the user to manually label individual warnings. Warnings that correspond to real flaws can be labeled as true positives.

Warnings that are false alarms can be labeled as false positives. Warnings that are technically true positives but which are benign can be labeled as don’t care. Most tools offer features that allow the user to suppress reporting of such warnings in subsequent analyses.

2.3.2. Limitations of Static Analysis

The obvious limitation of static analysis is that most nontrivial program properties are undecidable. As a result, any static analysis technique must approximate the behaviour of the input program. For this reason, no static analysis technique can be both complete and sound. In the literature on using program analysis to find bugs, sound generally means “finds every real bug”, and complete generally means “reports only real bugs.”

Deciding how to approximate in a static analysis has important consequences. Both soundness and completeness are desirable properties. In theory, a sound analysis is able to find every real instance of the kind of bugs the analysis is designed to detect. However, it might do this by reporting 1,000 false positives for every accurate warning, making use of the analysis unproductive. Similarly, a complete analysis would only report definite bugs. However, it might find only a very small number of bugs, leaving a large number of false negatives. For these reasons, tools which are neither complete nor sound can serve a valuable role in the software quality assurance process as long as they find a significant number of real bugs without emitting too many false positives.

In order to understand the limitations of the techniques that these tools use, it is important to understand the metrics used to assess their performance. The first metric, *recall*, is a measure of the ability of the tool to find real problems. Recall is measured as the number of flaws found divided by all flaws present. The second metric is *precision*, which measures the ability of the tool to exclude false positives. It is the ratio of true positives to all warnings reported. The third metric is *performance*. Although not formally defined, this is a measure of the computing resources needed to generate the results.

These three metrics usually operate in opposition to each other. It is easy to create a tool that has perfect precision and excellent performance, one that reports no lines contain flaws will satisfy because it reports no false positives. Similarly, it is easy to create a tool with perfect recall and excellent performance, one that reports that all lines have errors will answer because it reports no false negatives. Clearly, however, neither tool is of any use whatsoever.

Finally, it is at least theoretically possible to write an analyzer that would have excellent precision and excellent recall given enough time and access to enough processing power. Whether such a tool would be as useless as the previous two example tools is debatable and would depend on just how much time it would take. What is clear is that no such tools currently exist and to create them would be very difficult.

As a result, all tools occupy a middle ground around a sweet spot that developers find most useful. Developers expect analyses to complete in time roughly proportional to the size of their code base and within hours rather than days. Tools that take longer simply do not get used because they take too long. Low precision means more false positives, which has an insidious effect on users. As precision goes down, even true positive warnings are more likely to be erroneously judged as false positives because the users lose trust in the tool.

For most classes of flaws, precision less than 80 percent is unacceptable. For more serious flaws however, precision as low as five percent may be acceptable if the code is to be deployed in very risky environments. It is difficult to quantify acceptable values for recall as it is impossible to measure accurately in practice, but clearly users would not bother using these tools at all if they did not find serious flaws that escape detection by other means.

Each of these constraints introduces its own set of limitations, however they are all interrelated. The reasons that lead to low recall are explained in more detail in the following sections.

2.3.2.1. Path Limitations

These analyses are path sensitive and this improves both recall and precision and is probably the key aspect of these products that makes them most useful. A full exploration of all paths through the program would be very expensive. If there are *n* branch points in a procedure, and there are no loops in that procedure, then the number of intraprocedural paths through that procedure can be as many as *2 n*. In practice, this is fewer because some branches are correlated, but the asymptotic behavior remains. If procedure calls and returns are taken into account, the number of paths is *doubly* exponential, and if loops are taken into account then the number of paths is unbounded. Clearly it is not possible for a tool to explore all of these paths. The tools restrict their exploration in two ways. First, loops are handled by exploring a small fixed number of iterations: often, the first time around the loop is singled out as special, and all other iterations are considered en masse and represented by an approximation. Second, not all paths are explored. It is typical for an analysis to place an upper bound on the number of paths explored in a particular procedure or on the amount of time available, and a selection of those remaining paths are explored.

If asynchronous paths occur (such as those caused by interrupts or exceptions) or if the program uses concurrency, then the number of possible paths to consider can increase further. Many tools simply ignore these possibilities. Finally, most tools also ignore recursive function calls, and function calls that are made through function pointers (or make very coarse approximations) as considering these also contributes to poor performance and poor precision.

2.3.2.2. Abstract Domain

As previously mentioned, these tools work by exploring paths and looking for anomalies in the abstract state of the program. The appeal of the symbolic execution is that each abstract state represents potentially many possible concrete states. For example, given an 8-bit variable x, there are 2^8 possible concrete values: 0, 1, …, 255. The symbolic execution, however, might represent the value as two abstract states: $x=0$, and $x>0$. So where a concrete execution has 256 states to explore, the symbolic execution has only two.

As such, the expressivity of this abstract domain is an important factor that determines the effectiveness of the analysis. Again, there is a trade-off here: better precision and recall can be achieved by more sophisticated abstract domains, but more resources will then be required to complete an analysis. Values in the abstract domain are equations that represent constraints on values, i.e., $x=0$, or $y>10$. As the analysis progresses, a constraint solver is used to combine and simplify these equations. A key characteristic of these abstract domains is that there is a special value, usually named *bottom*, which indicates that the analysis knows no useful information about the actual value. *Bottom* is the abstract value that corresponds to all possible concrete values. Reaching bottom is impossible to avoid for any non-trivial abstraction in general as this would require solving the halting problem. Once bottom is reached, the analysis has a choice of treating it as a potentially dangerous value, which would increase recall, or as a probably safe value, which would increase precision. Most tools opt for the latter as the former also has the effect of decreasing precision enormously.

If there are program constructs that step outside the bounds of what can be expressed in the abstract domain, this causes the analysis to lose track of variables and their relationships. For example, an abstract domain that allows the expression of affine relationships between no more than two variables admits expressions such as $x=2y$. However, something such as $x=y+z$ is out of bounds because it involves three variables and the analysis would be forced to conclude *x=bottom* instead.

The consequence of this is the abstract domain that a tool uses determine a great deal about the kind of flaws that it is capable of detecting. For example, if the tool uses an abstract domain of affine relations between two variables, then it may fail to find flaws that depend on three variables. Similarly, most tools choose a domain that allows them to reason about the values of integers and addresses but not floating-point values, so they will fail to find flaws in floating-point arithmetic (such as divide by zero).

2.3.2.3. Missing Source Code

If the source code to a part of a program is not available, as is almost always the case because of operating system and third-party libraries, or if the code is written in a language not recognized by the analysis tool, then the analysis must make some assumptions about how that missing code operates. Take, for example, a call to a function in a third-party library that takes a single pointer-typed parameter and returns an integer. In the absence of any other information, most analyses will assume that the function does nothing and returns an unknown value. This clearly is not realistic, but it is not practical to do better in general. The function may de-reference its pointer parameter, it may read or write any global variable that is in scope, it may return an integer from a particular range, or it may even abort execution. If the analysis knew this, it would have better precision and recall but it is forced to make the simple assumption unless told otherwise.

There are two approaches around this. First, if source is not available but object code is, then the analysis could be extended into the object code. This is a highly attractive solution but no products are available yet. The technological basis for such a tool exists, however (Balakrihnan et al., 2005), and it is expected that products capable of analyzing object code as well as C/C++ will appear.

A second approach to the problem is to specify stubs, or *models*, that summarize key aspects of the missing source code. The popular analysis tools provide models for commonly used libraries such as the C library. These models only have to approximate the behavior of the code. Users can write these themselves for their own libraries but it can be a tricky and time-consuming effort.

2.3.2.4. Out of Scope

There are entire classes of flaws that static analysis is unlikely ever to be able to detect. Static analysis excels at finding places where the fundamental rules of the language are being violated such as buffer overruns, or where commonly used libraries are being used incorrectly, or where there are inconsistencies in the code that indicate misunderstanding. If the code does the wrong thing for some other reason, but does not then terminate abnormally, then static analysis is unlikely to be able to help because it is unable to divine the intent of the author. For example, if a function is intended to sort in ascending order, but perfectly sorts in descending order instead, then static analysis will not help much. This kind of functionality testing is what traditional dynamic testing is good for.

2.3.3. Open Source Tools

Open source software development is based on a relatively simple idea: the core of the system is developed locally by a single programmer or a team of programmers. A prototype system is released on the Internet, which other programmers can freely read, modify and redistribute the system's source code. The evolution of the system happens in an extremely rapid way; much faster than the typical rate of a 'closed' project. It appears that open source is presenting the traditional software development industry with an important challenge (Stamelos et al., 2002). SourceForge.net, the world's largest open source development website, hosts over 70,000 open-source projects for more than 700,000 registered developers (Huang et al., 2004).

It is often claimed that open source software is intrinsically more secure than closed source or commercial software. Others argue that it is not, and it is expected this debate will continue for some time to come. The availability of source code provides both attackers and defenders opportunities to study code in detail and identify software vulnerabilities. On the other hand, closed source software forces users to accept only the level of security diligence that the vendor chooses to provide (Covan, 2003). According to the Open Source Initiative (opensource.org, 2010), the terms for the distribution of open source software must also comply with 10 criteria specified in the Open Source Definition (opensource.org, 2010). The top 3 items out of the 10 criteria include:

1. software should be freely redistributable,
2. software must allow for distribution as source code as well as in a compiled form,
3. licences must allow modifications and for derivatives generated from the source code.

In Appendix B, several open source static analysis tools included in Multi-language, NET (C#, VB.NET and all .NET compatible languages), Java, C, C++, Objective-C, Perl and ActionScript are explained.

2.3.4. Commercial Tools

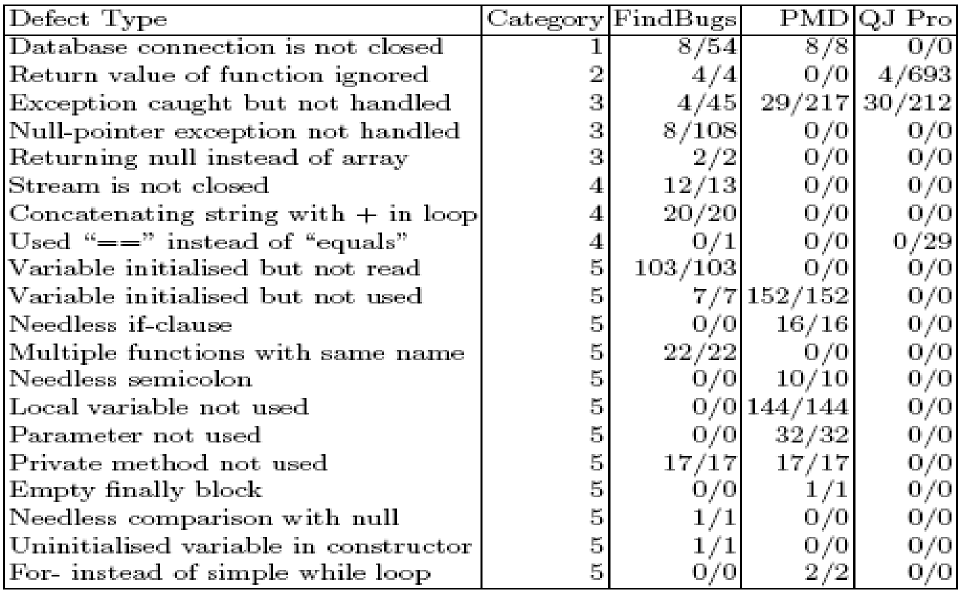
One distinct difference between open source and commercial software is the availability of source code for review. Commercial software is mostly closed source where the source code is not publicly available. Because the source code is not available, there is a barrier against access to the code that attackers have to cross, resulting in less likelihood of vulnerabilities in the source code being exploited even though vulnerabilities do exist (Covan, 2003). In Appendix C, commercial static analysis tools are summarized.

2.3.5. Comparison of Static Analysis Tools for Java

Static analysis tools for Java analysis code are compared in Table 2.2. Table 2.2 indicates the defect types with their categories and the corresponding positives found by each tool. The number before the slash denotes the number of true positives, the number after the slash the number of all positives.

Most of the true positives can be assigned to the category (maintainability of the code). It is noticeable that the different tools predominantly find different positives. Only a single defect type was found from all tools, four types from two tools each. Considering the categories, FindBugs finds in the different systems positives from all categories and PMD only from the categories Failure of the application, *Insufficient error handling*, and *Maintainability of the code*. QJ Pro only reveals positives from the categories *Logical failure of the application*, *Insufficient error handling*, and *Violation of structured programming*. The number of faults found in each category from each tool is graphically illustrated in Figure 2.6. Also the number of types of defects varies from tool to tool. FindBugs detects defects of 13 different types, PMD of 10 types, and QJ Pro only of 4 types.

Table 2.2. Comparison of static analysis tools
(Source: Wagner et al., 2005)



Defect Type	Category	FindBugs	PMD	QJ Pro
Database connection is not closed	1	8/54	8/8	0/0
Return value of function ignored	2	4/4	0/0	4/693
Exception caught but not handled	3	4/45	29/217	30/212
Null-pointer exception not handled	3	8/108	0/0	0/0
Returning null instead of array	3	2/2	0/0	0/0
Stream is not closed	4	12/13	0/0	0/0
Concatenating string with + in loop	4	20/20	0/0	0/0
Used “==” instead of “equals”	4	0/1	0/0	0/29
Variable initialised but not read	5	103/103	0/0	0/0
Variable initialised but not used	5	7/7	152/152	0/0
Needless if-clause	5	0/0	16/16	0/0
Multiple functions with same name	5	22/22	0/0	0/0
Needless semicolon	5	0/0	10/10	0/0
Local variable not used	5	0/0	144/144	0/0
Parameter not used	5	0/0	32/32	0/0
Private method not used	5	17/17	17/17	0/0
Empty finally block	5	0/0	1/1	0/0
Needless comparison with null	5	1/1	0/0	0/0
Uninitialised variable in constructor	5	1/1	0/0	0/0
For- instead of simple while loop	5	0/0	2/2	0/0

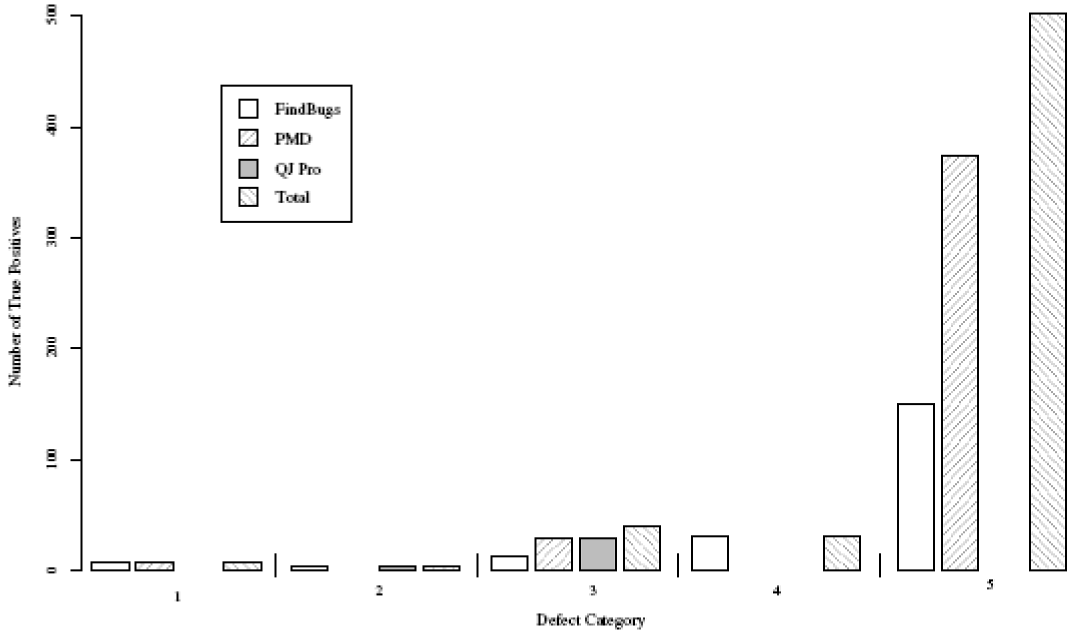


Figure 2.6. A graphical comparison of number of true positives for each tool (Source: Wagner et al., 2005)

The accuracy of the tools is also diverse. Using the defect type “Exception is caught but not handled” that can be found by all three tools as an example. While FindBugs only finds 4 true positives, PMD reveals 29 and QJ Pro even 30. For this, the result from QJ Pro contains the true positives from PMD which in turn contain the ones from FindBugs. A reason for this is that QJ Pro is also able to recognize a single semicolon as a non-existent error handling, whereas the other two interpret that as a proper handling. This defect type is also representative in the way that FindBugs finds the least true positives. This may be the case because it uses the compiled class-files while PMD and QJ Pro analyse the source code.

CHAPTER 3

PMD

3.1. Overview of PMD

PMD is an open-source, rule based, static source code analyzer that analyzes Java source code based on the evaluative rules that have been enabled during a given execution. It finds potential problems in Java source code, unused variables, empty catch blocks, wasteful object creation. Unlike a runtime analysis tool such as a memory profiler, a "static" code analysis tool such as PMD finds problems without actually executing code. PMD comes with more than 140 built-in rules to check source code. It also contains a framework that allows writing rules, so it can be checked project's source code for problems specific to environment (Copeland, 2005). The tool comes with a default set of rules which can be used to unearth common development mistakes such as having empty try-catch blocks, variables that are never used, objects that are unnecessary etc. Additionally, PMD also allows users to execute custom analyses by allowing them to develop new evaluative rules in a convenient manner. The tool comes with a relatively easy to user command line interface and at the same time, can also be integrated with popular development environments such as Eclipse. It is also a fact that PMD is quite popular in the Java development community and has also achieved substantial industry acceptance. Currently it only supports Java and as can be ascertained from the second section of this document, while there is room for improvement, the tool is generally quite effective and has proven itself as a useful static analysis tool for both large and small code bases.

PMD operates in a manner that is very similar to conventional static analysis tools. This naturally means that the tool involves the generation and traversal of an abstract syntax tree. In summary, based on the information obtained from PMD project's official website at sourceforge.com, PMD's internal operation can be summarized as follows (Hsu et al., 2007):

(1) User supplies the location of the source code that has to be analyzed along with the rule or rules that it would be to execute;

(2) The tool opens a data read stream to read in the source code and supplies it to a Java based code parser which in turn generates an Abstract Syntax Tree (AST) (SourceForge.net-a,2010);

(3) The AST is then returned to PMD which in turn gives it to the “symbol table layer” that identifies scopes, declarations, and various usages;

(4) If a particular rule involves data flow analysis, the AST is given by PMD to the deterministic finite automaton (DFA) layer that in turn generates control flow graphs and data flow nodes;

(5) With all this data obtained, each rule traverses the abstract syntax tree as needed and detects issues based on this traversal, rules can also utilize symbol tables and nodes within the generated DFA;

(6) The issues identified in step 5 are then printed out to the console or an associated file in a number of different formats.

3.2. Rulesets of PMD

PMD uses rulesets to evaluate code. A rule property is similar to a parameter. It allows changing the rule’s behaviour without changing the rule itself. The applied rulesets and the results observed are listed in Appendix C (SourceForge.net-a, 2010).

3.3. Writing Rules

PMD has two general mechanisms for creating rules (SourceForge.net-a, 2010); XPath and creating a Java class. XPath (for XML Path Language) is a domain-specific language used for locating parts of an XML document. It’s also a World Wide Web Consortium (W3C®).

When using XPath to write a PMD rule, specify an XPath expression to select the code that it is wanted to flag as a violation. XPath rules can be quick and easy to write, but it is limited to built-in XPath functionality, no have access to the full power of the Java programming language. Even with this limitation, however, XPath is an excellent prototyping mechanism and, for many rules, may be powerful enough to do the trick.

When using Java to write a rule, write a Java class that extends the net.sourceforge.pmd.AbstractRule abstract base class and then fill in code to detect problems. Writing rules in Java requires a compilation step between writing the rule and running the code, so it takes a bit more labour. But Java rules are very flexible and powerful, and can execute much faster than XPath rules.

3.3.1. Writing XPath Rules

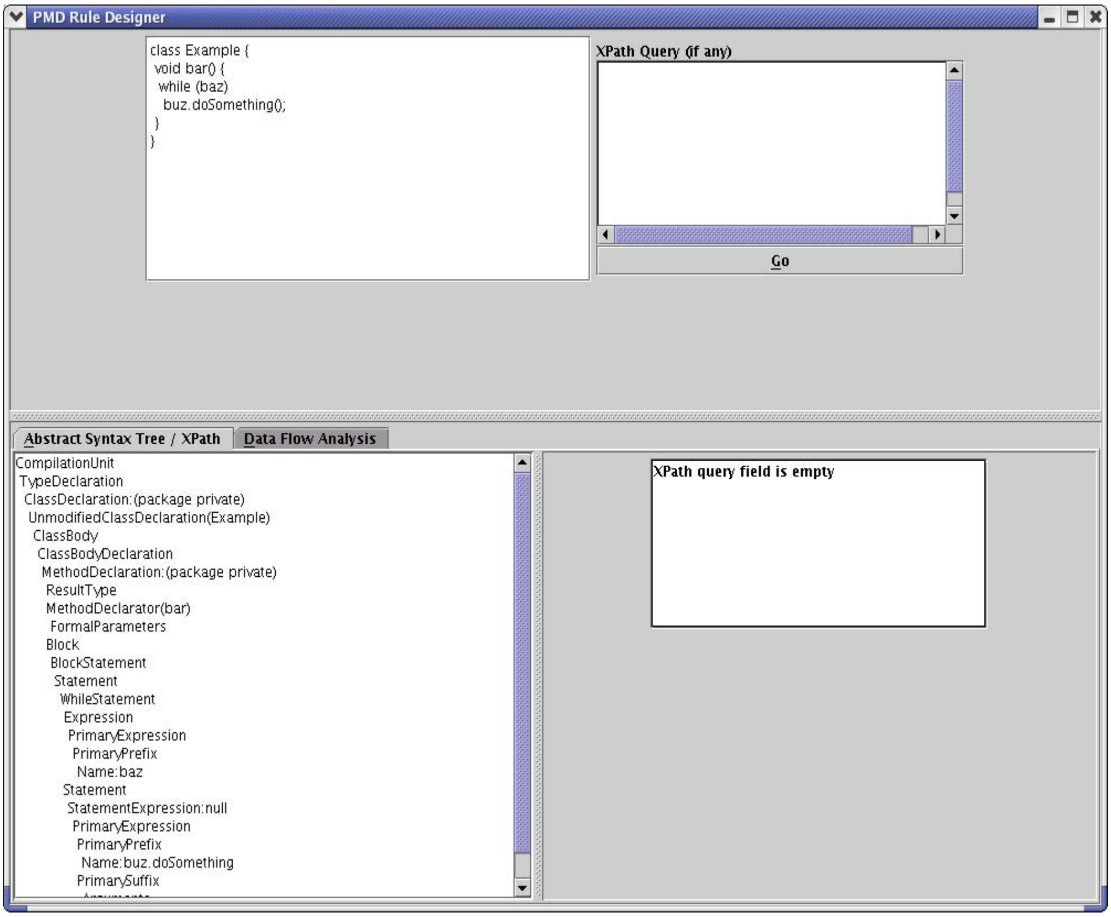
PMD doesn't use the source code directly; it uses a JavaCC generated parser to parse the source code and produce an AST (Abstract Syntax Tree). For example, if the code is:

```
class Example {
 void bar() {
  while (baz)
   buz.doSomething();
 }
}
```

the AST for the code above looks like this:

```
CompilationUnit
 TypeDeclaration
  ClassDeclaration:(package private)
   UnmodifiedClassDeclaration(Example)
    ClassBody
     ClassBodyDeclaration
      MethodDeclaration:(package private)
       ResultType
       MethodDeclarator(bar)
        FormalParameters
       Block
        BlockStatement
         Statement
          WhileStatement
           Expression
 PrimaryExpression
  PrimaryPrefix
   Name:baz
Statement
 StatementExpression:null
  PrimaryExpression
   PrimaryPrefix
    Name:buz.doSomething
   PrimarySuffix
    Arguments
```

It can be also generated the AST by using PMD's Designer Utility which is included in PMD software as shown in Figure 3.1.



```
class Example {
 void bar() {
  while (baz)
   buz.doSomething();
 }
}
```

```
CompilationUnit
 TypeDeclaration
  ClassDeclaration:(package private)
   UnmodifiedClassDeclaration(Example)
    ClassBody
     ClassBodyDeclaration
      MethodDeclaration:(package private)
       ResultType
       MethodDeclarator(bar)
        FormalParameters
       Block
        BlockStatement
         Statement
          WhileStatement
           Expression
            PrimaryExpression
             PrimaryPrefix
              Name:baz
           Statement
            StatementExpression:null
             PrimaryExpression
              PrimaryPrefix
               Name:buz.doSomething
              PrimarySuffix
```

XPath query field is empty

Figure 3.1. PMD Rule Designer.

Recently Daniel Sheppard enhanced PMD to allow rules to be written using XPath. XPath is a way of querying an XML document. It can be written an XPath query to get a list of nodes that fit a certain pattern. Daniel Sheppard downloaded the Jaxen XPath engine (Codehous.org, 2010) and wrote a class called a DocumentNavigator that allows Jaxen to traverse the AST. Jaxen gets the XPath expression, evaluates it, applies it to the AST, and returns a list of matching nodes to PMD. PMD creates RuleViolation objects from the matching nodes and moves along to the next source file.

For example, the XPath expression for “While Loops Must Use Braces Rule” looks like this:

```
//WhileStatement[not(Statement/Block)]
```

It is needed that XPath rules are set the class attribute in the rule definition to net.sourceforge.pmd.rules.XPathRule. The example of complete XPath rule looks as follows:

```
<rule name=" WhileLoopsMustUseBracesRule"
        message="Avoid while loops without braces"
    class="net.sourceforge.pmd.rules.XPathRule">
    <description>
    etc., etc.
<rule name=" WhileLoopsMustUseBracesRule "
    message="Don't Avoid while loops without braces"
    class="net.sourceforge.pmd.rules.XPathRule">
  <description>
      …
  </description>
  <properties>
  <property name="xpath">
    <value>
    <![CDATA[
      //WhileStatement[not(Statement/Block)]
    ]]>
    </value>
  </property>
  </properties>
  <example>
        <![CDATA[
                While(..)
                        Code line // don't do this!
                ]]>
  </example>
 </rule>
```

3.3.2. Writing Java Rules

Writing rules using XPath expressions is quick and straightforward. It can be used the Designer to experiment with various XPath expressions. Writing rules in Java has some real advantages:

1. Having full power of the Java programming language at disposal. It is not limited to the built-in XPath functions; it can be used StringTokenizer, collection classes, regular expressions, or anything else.
2. Java rules run faster. This is an unavoidable result of using a domain specific language like XPath; before the XPath rule can be used it needs to be parsed and interpreted by the XPath engine. This extra layer of indirection doesn't come free.

There is plenty of common ground between writing rules in XPath and rules in Java. Usually it can be spot the problem in the AST, whether it's a simple sequence of nodes or a particular node attribute. Once the Java rule is written, it is needed to package it inside a ruleset just as in the XPath rule (although the XML specification is a bit different).

Generally, a Java rule consists of a Java class that extends the net.sourceforge.pmd.AbstractRule. When PMD is run, it will traverse the AST and notify rule when it finds a node type for which rule implements a callback method. For example, if to check field names, it can be written the rule to receive a callback whenever the AST traversal finds a FieldDeclaration node. Design pattern aficionados will recognize this as a classic Visitor pattern; it's also known as "double dispatch".

A series of node type imports. These could be reduced to one line since they all come from the same package,

```
import net.sourceforge.pmd.ast.ASTIfStatement;
import net.sourceforge.pmd.ast.ASTStatement;
import net.sourceforge.pmd.ast.ASTBlock;
import net.sourceforge.pmd.ast.ASTEmptyStatement;
import net.sourceforge.pmd.ast.Node;
```

To bring in imports of PMD infrastructure classes; AbstractRule is extended and RuleContext can be used to store the rule violations:

```
import net.sourceforge.pmd.AbstractRule;
import net.sourceforge.pmd.RuleContext;
```

Now the rule class itself:

```
public class EmptyIf extends AbstractRule {
```

Callback method will be invoked every time an IfStatement node occurs in a source file having PMD check. The first parameter, ASTIfStatement, identifies the node that's being visited. The second parameter, the Object**,** is really an instance of RuleContext. Notice that the node class names are preceded with "AST", so instead of an IfStatement class name it's ASTIfStatement.

```
public Object visit(ASTIfStatement node, Object data) {
```

An IfStatement node is guaranteed to have a Statement node as its second child node. Since node indexes are zero based, it can be get a handle to it by invoking jjtGetChild(1) on the IfStatement node. Note that some methods have a 'jjt' prefix; that's because they are methods generated by the JJTree utility that comes with JavaCC. Since jjtGetChild returns a generic Node interface type,

```
ASTStatement stmt = (ASTStatement)node.jjtGetChild(1);
```

It can be either a Block or an EmptyStatement. So to leave it as a Node type:

```
Node stmtChild = stmt.jjtGetChild(0);
```

Analyze the node to see what it is. In the XPath expression, [EmptyStatement] is used to check for an EmptyStatement child; here the node type will be checked with an instanceof keyword:

```
if (stmtChild instanceof ASTEmptyStatement) {
```

If it is found an EmptyStatement, it is needed to flag this IfStatement node as a problem. Since it is extended AbstractRule, it can be used a utility method, addViolation, to add a new rule violation. Data parameter that is really a RuleContext object reference is passed;

```
addViolation(data, node);
```

That is the EmptyStatement case. The other case is a bit more complicated, if the child node is a Block, it is needed to ensure that it has no children by using the instanceof keyword and the jjtGetNumChildren function. This function returns the number of children that are attached to the node. In the following, a rule violation occurs if the child node is a block *and* the block has children:

```
} else if (stmtChild instanceof ASTBlock
            && stmtChild.jjtGetNumChildren() == 0) {
addViolation(data, node);
}
```

After checking two possibilities, it can be continued the traversal. Invoking super.visit(node, data) will do the trick:

```
return super.visit (node, data);
```

the entire source file for this rule as follows:

```
package net.sourceforge.pmd.rules;
import net.sourceforge.pmd.ast.ASTIfStatement;
import net.sourceforge.pmd.ast.ASTStatement;
import net.sourceforge.pmd.ast.ASTBlock;
import net.sourceforge.pmd.ast.ASTEmptyStatement;
import net.sourceforge.pmd.ast.Node;
import net.sourceforge.pmd.AbstractRule;
import net.sourceforge.pmd.RuleContext;
public class EmptyIf extends AbstractRule {
public Object visit(ASTIfStatement node, Object data) {
  ASTStatement stmt = (ASTStatement)node.jjtGetChild(1);
  Node stmtChild = stmt.jjtGetChild(0);
  if (stmtChild instanceof ASTEmptyStatement) {
   addViolation(data, node);
   } else if (stmtChild instanceof ASTBlock &&
stmtChild.jjtGetNumChildren() == 0) {
   addViolation(data, node);
   }
   return super.visit(node, data);
   }
   }
```

As with the XPath rule, it is needed to wrap the rule in a new ruleset file. There are two differences between XPath and Java rules. First, an XPath rule has a class attribute in the rule element that point to net.sourceforge.pmd.rules.XPathRule. For a Java rule, that attributes points to the new class. Second, XPath requires a properties element; Java does not, which makes for a shorter rulesets.

CHAPTER 4

PMD RULES FOR SECURITY VULNERABILITIES

4.1. Overview of Security Vulnerabilities

Vulnerability is a set of conditions that allows violation of an explicit or implicit security policy. Programs, systems, and networks exhibit vulnerabilities (Secord and Householder, 2005). Vulnerability is also defined as a state of the system from which it is possible to transition to an incorrect system state. In other words, vulnerability is a software system defect which, when exercised, can produce undesirable or incorrect behaviour. In contrast, an exploit is the *process* by which one or more vulnerabilities are exercised to attack a system (Bazaz et al., 2006).

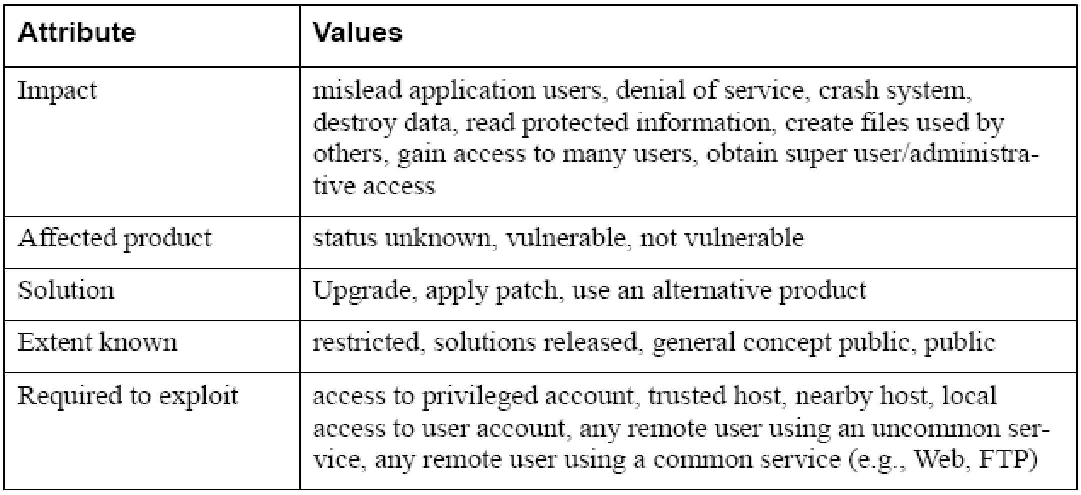
Because vulnerabilities are central to exploiting a software application, one can prevent an exploit by identifying, and subsequently eliminating, vulnerabilities present in a software application. However, identifying if and which vulnerabilities are present have been affected several difficulties including;

1. *The complexity of software applications:* Modern software applications are often large, complex, and contain thousands of lines of code. Furthermore, application complexity increases with the number of services it uses which are provided by the other applications.

2. *The number of potential vulnerabilities:* Because numerous vulnerabilities exist, attempting to identify the specific one(s) present in a software application from a list of possibilities is impractical.

3. *The complexity of vulnerabilities:* Some vulnerabilities, such as those used in the “time of check, time of use” exploit, involve multiple software components interacting together to produce the vulnerable system state. This introduces additional layers of complexity. Table 4.1 shows the properties of vulnerabilities with associated values.

Table 4.1. Properties of vulnerabilities
(Source: Secord and Householder, 2005)



Attribute	Values
Impact	mislead application users, denial of service, crash system, destroy data, read protected information, create files used by others, gain access to many users, obtain super user/administrative access
Affected product	status unknown, vulnerable, not vulnerable
Solution	Upgrade, apply patch, use an alternative product
Extent known	restricted, solutions released, general concept public, public
Required to exploit	access to privileged account, trusted host, nearby host, local access to user account, any remote user using an uncommon service, any remote user using a common service (e.g., Web, FTP)

Vulnerabilities have been classified into broad categories such as buffer overflows, format string vulnerabilities, and integer type range errors (including integer overflows). These broad categories however have two major failings. First, it is not always possible to assign a vulnerability to a single category. Second, the distinctions are too general to be useful in any detailed engineering analysis.

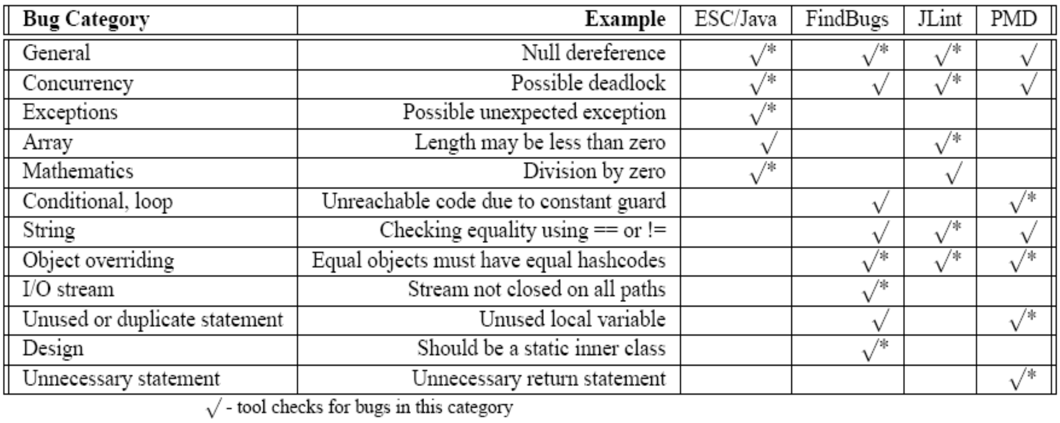
For example, the following function:

```
bool func(char *s1, int len1,
char *s2, int len2) {
char buf[128];
if (1 + len1 + len2 > 128) return false;
if (buf) {
strncpy(buf, s1, len1);
strncat(buf, s2, len2);
}
return true;
}
```

contains a vulnerability in that len1 or len2 could be a negative number, allowing the length check to be bypassed but still causing a buffer overflow in the strncpy() or strncat() functions. Is this integer range value vulnerability because the integer range check was bypassed, or is this simply a buffer overflow? Either categorization would be a disservice to understanding the issues.

Understanding vulnerabilities is critical to understanding the threats they represent. Classification of vulnerabilities allows collection of frequency data and trend analysis of vulnerabilities but has not been regularly or consistently applied. Better and more comprehensive classification of vulnerabilities can lead to better correlation with incidents, exploits, and artifacts and can be used to determine the effectiveness of countermeasures. Understanding the characteristics of vulnerabilities and exploits is also essential to the development of a predictive model that can predict threats with a high correlation and significance. Table 4.2 shows the classification of vulnerabilities. The first column lists a general class of bugs, and the second column gives one common example from that class. The last columns indicate whether each tool finds bugs in that category, and whether the tools find the specific example.

Table 4.2. Types of vulnerabilities for each tool finds
(Source: Rutar et al., 2004)



Bug Category	Example	ESC/Java	FindBugs	JLint	PMD
General	Null dereference	√*	√*	√*	√
Concurrency	Possible deadlock	√*	√	√*	√
Exceptions	Possible unexpected exception	√*			
Array	Length may be less than zero	√		√*	
Mathematics	Division by zero	√*		√	
Conditional, loop	Unreachable code due to constant guard		√		√*
String	Checking equality using == or !=		√	√*	√
Object overriding	Equal objects must have equal hashcodes		√*	√*	√*
I/O stream	Stream not closed on all paths		√*		
Unused or duplicate statement	Unused local variable		√		√*
Design	Should be a static inner class		√*		
Unnecessary statement	Unnecessary return statement				√*

√ - tool checks for bugs in this category

4.2. PMD Rules for Security Vulnerabilities

PMD includes checks for some common bug patterns, such as the well-known double-checked locking bug in Java. Like PMD, FindBugs also checks for uses of double checked locking. PMD does not check for null pointer dereferences, but it does warn about setting certain objects to null. PMD does not check for array bounds errors, though FindBugs does warn about returning null from a method that returns an array.

ESC/Java includes support for automatically checking for race conditions and potential deadlocks. ESC/Java reports synchronized blocks that are involved in potential deadlocks but not the sets of locks in each particular deadlock. ESC/Java reports the most null pointer dereferences because it often assumes objects might be null. In Java, indexing outside the bounds of an array results is a run-time exception. While a bounds error in Java may not be the catastrophic error that it can be for C and C++ (where bounds errors overwrite unexpected parts of memory), they still indicate a bug in the program. JLint and ESC/Java, include checks for array bounds errors, either creating an array with a negative size, or accessing an array with an index that is negative or greater than the size of the array. ESC/Java mainly reports warnings because parameters that are later used in array accesses may not be within range.

FindBugs warns about the presence of other concurrency bug patterns, such as not putting a monitor wait() call in a while loop. The warnings FindBugs reports indicate the presence of the bug pattern in the code. What is less clear is how many of the patterns detected correspond to actual errors. For example, since FindBugs does not perform interprocedural analysis (it analyzes a single method at a time), if a method with a wait() is itself called in a loop, FindBugs will still report a warning. FindBugs discovers a very small set of potential null dereferences compared to both ESC/Java and JLint. This is because FindBugs uses several heuristics to avoid reporting null-pointer dereference warnings in certain cases when its dataflow analysis loses precision.

JLint generates many warnings about potential deadlocks. In some cases, JLint produces many warnings for the same underlying bug. For instance, JLint checks for deadlock by producing a lock graph and looking for cycles. JLint iterates over the lock graph repeatedly. Among the four tools, ESC/Java, FindBugs, and JLint check for null dereferences. JLint finds many potential null dereferences. In order to reduce the number of warnings, JLint tries to only identify inconsistent assumptions about null. For example, JLint warns if an object is sometimes compared against null before it is dereferenced. JLint has several false positives and some false negatives in array bounds errors, because it does not track certain information interprocedurally in its dataflow analysis.

4.3. Boundary Overflow Vulnerabilities

Boundary overflows are caused by violation of constraints, mostly limiting the range of internal values of program, and can be provoked by an intruder to gain control of or access to stored data (Tuglular et al., 2009a). Boundary overflow vulnerability is characterized as a boundary overflow when the input being received by a system, whether human or machine-generated, causes the system to exceed an assumed boundary, thereby causing vulnerability. For example, the system may run out of memory, disk space, or network bandwidth. Another example is that a variable might reach its maximum value and roll over to its minimum value and variables in an equation might be set such that a division by zero error occurs is the third example. Boundary overflow errors are a subset of the class of input validation errors (Mell and Tracy, 2002).

Boundaries cause overflow vulnerabilities are not related to the maximum length of the variables; however, they are predefined values that are related to the semantics of the graphical user interface (GUI) elements. For example, consider port scanners where the start and end port values are entered to the program to scan the range of these values. Even if the input values that are entered from the GUI elements for the start and end port values don't exceed the maximum length of the unsigned integer type, there are semantically predefined boundary values. The start port value can not be lower than 0 and the end port value can not be higher than 65535. Checking these input values with respect to the redefined boundaries is critical for the program to behave as intended. Otherwise, boundary overflow occurs and the program works in an unexpected way (Muftuoglu, 2009).

CHAPTER 5

TOOL SUPPORT AND CASE STUDY

5.1. Boundary Overflow Vulnerability Checker Tool

In this thesis, a detection algorithm which is based on the notion of static analysis by using PMD is implemented to check the overflow vulnerabilities. The implemented detection algorithm is capable of finding any type of vulnerability which can be represented as Boolean expressions.

Boundary Overflow Vulnerability Checker is a tool developed in Java by using Eclipse Version 3.4.2 IDE on Intel machine with 32bit architecture. It uses PMD structure for static analysis in background. It allows users to choose a target source codes folder and define input contracts to analyse for boundary overflow vulnerability. It scans for defined input contracts in selected folder and writes results to the screen.

5.1.1. Design and Implementation

The algorithm scans the source code statically, finds and shows local variables, method/constructor parameters to define input contracts easily. Once input contracts are defined, it searches and writes occurrences of the input contracts if exists. It also warns that if the input contact was not found.

Boundary overflow checker algorithm consists of four steps. In step 1, target folder which contains java source codes is selected. Java file chooser (JFileChooser) component was used for the folder selection process. File chooser provides a GUI for navigating the file system, and then choosing a directory from a list, or entering the name of a directory.

In step 2, the local variable and method/constructor parameter definitions along with their specified types are tracked from the source codes (which are in selected folder in step 1) and displayed in separated lists. The type of variables was limited by integer type. It is possible to show all type of variables but displaying all the variables in a program makes no sense if only some specified types are needed. A PMD java rule was written and integrated to find local variables and method/constructor parameters. In this step that rule was used internally.

In step 3, it is expected that user should define input contracts either by using local variables and method/constructor parameters which are mentioned in step 2. Input contracts are formed of:

- left operand: specifies the name of the variable/parameter or a numerical value
- operator: is the Boolean operator
- right operand: specifies the name of the variable/parameter or a numerical value

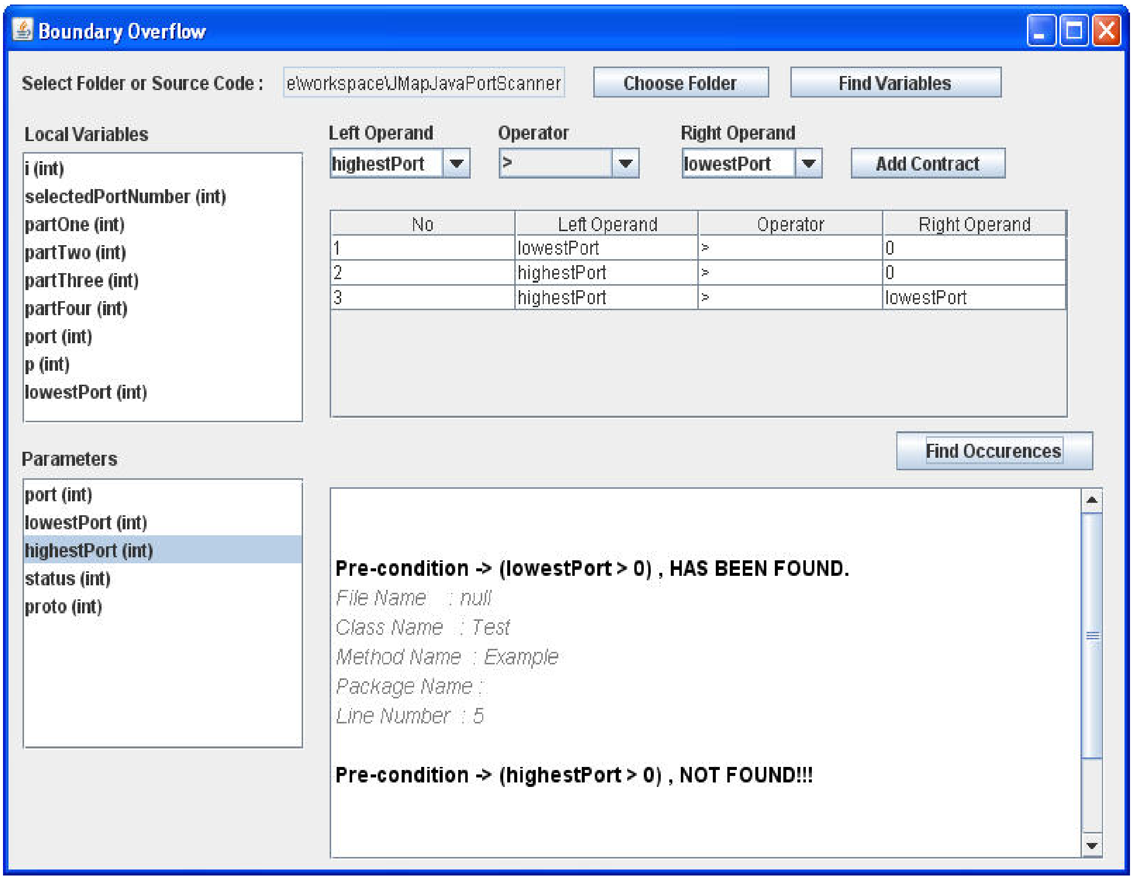
Left and right operands can be selected from the dropdown lists which contains all the name of found variables/parameters in step 2 or can be written by hand for numerical values.

Step 4 scans the source code(s) to check all input contracts separately. Input contracts are compared with the conditions of the variable written in the source code(s). If the condition of the variables matches with any of the input contracts, the input contract is satisfied by the source code and prints the occurrence of the condition to the bottom part of the tool.

5.1.2. Usage of the Tool

Figure 5.1 shows GUI of the boundary overflow vulnerability checker tool that allows to input source directory of the software to be checked, shows the lists of local variables and parameters, enables to write input contracts and displays the outputs.

As can be seen in the GUI of the boundary overflow vulnerability checker tool, the tool takes the source directory of the program. After selection of source directory, local variables and method/constructor parameters are loaded to the lists that are located on the left side of the GUI by clicking “Find Variables” button. After that at least one input contract has to be defined via left operand, operator, right operand dropdown lists. Adding new input contract updates the input contract table and inserts the new contract to the end of the table as a row. After definitions of the input contracts “Find Occurrences” button allows to start checking source code(s). The occurrences with file name, class name, method name, package name, line number information are printed to the bottom of the GUI as results. It is also printed that even if input contract does not found.



Select Folder or Source Code : e\workspace\JMapJavaPortScanner

Local Variables

i (int)
selectedPortNumber (int)
partOne (int)
partTwo (int)
partThree (int)
partFour (int)
port (int)
p (int)
lowestPort (int)

Left Operand: highestPort
Operator: >
Right Operand: lowestPort

No	Left Operand	Operator	Right Operand
1	lowestPort	>	0
2	highestPort	>	0
3	highestPort	>	lowestPort

Parameters

port (int)
lowestPort (int)
highestPort (int)
status (int)
proto (int)

```
Pre-condition -> (lowestPort > 0) , HAS BEEN FOUND.
File Name    : null
Class Name   : Test
Method Name  : Example
Package Name :
Line Number  : 5

Pre-condition -> (highestPort > 0) , NOT FOUND!!!
```

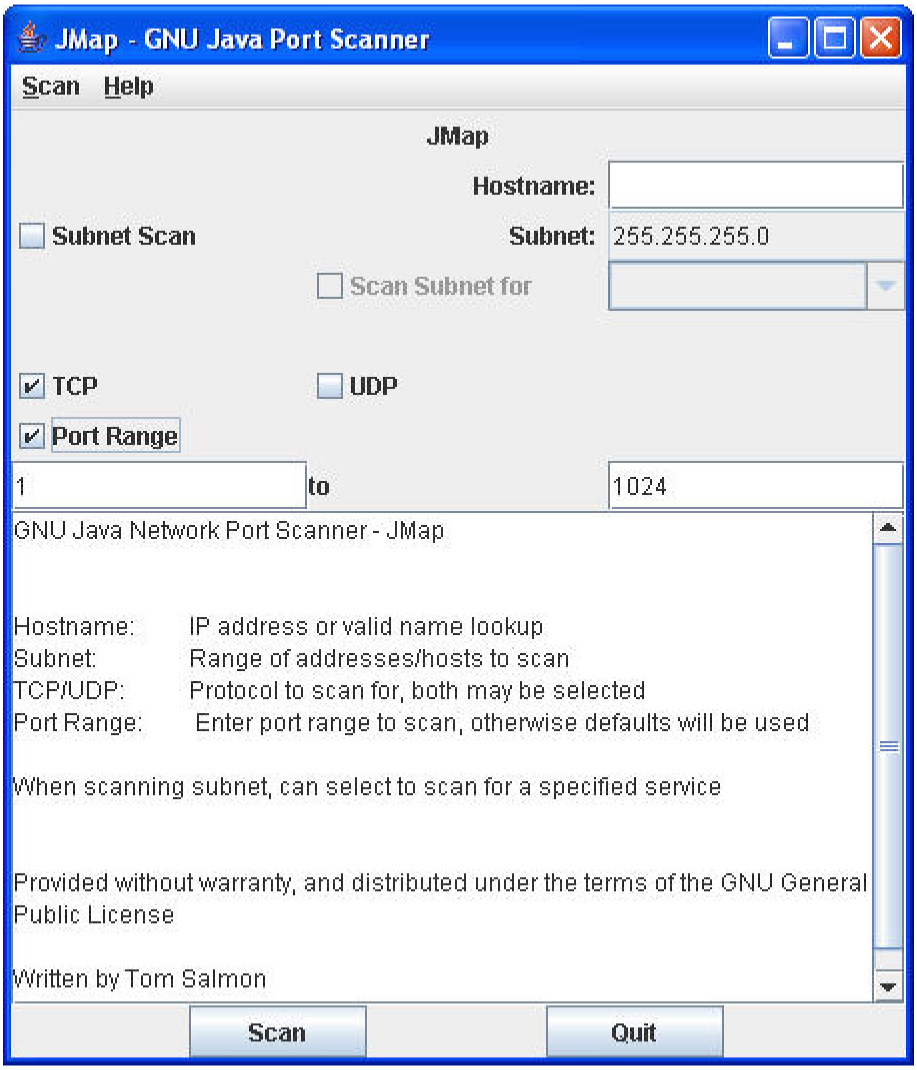
Figure 5.1. Boundary overflow vulnerability checker tool.

5.2. Case Study

The implementation and the tool introduced in Section 5.1.1 are evaluated in this section by using port scanner software. A port scan function scans a single port or a range of ports, i.e., ports between a given start and end, to check whether they are open or not. Test cases are generated for start and end port. Boundary overflow vulnerability analysis tool analyzed the source directory to detect the vulnerabilities related to boundary overflow.

The case study is performed on the basis of the port scanner part of open source port scanner software, i.e., JMAP Java Port Scanner (TomSalmon.com, 2010) and its

GUI is shown in Figure 5.2. Faltron Java Port Scanner (FaltronSoft.org, 2010) is also given in Figure 5.3. And a port scanner (Planet-Sourcecode.com, 2010), is shown in Figure 5.4.



Hostname:

☐ Subnet Scan Subnet: 255.255.255.0

☐ Scan Subnet for

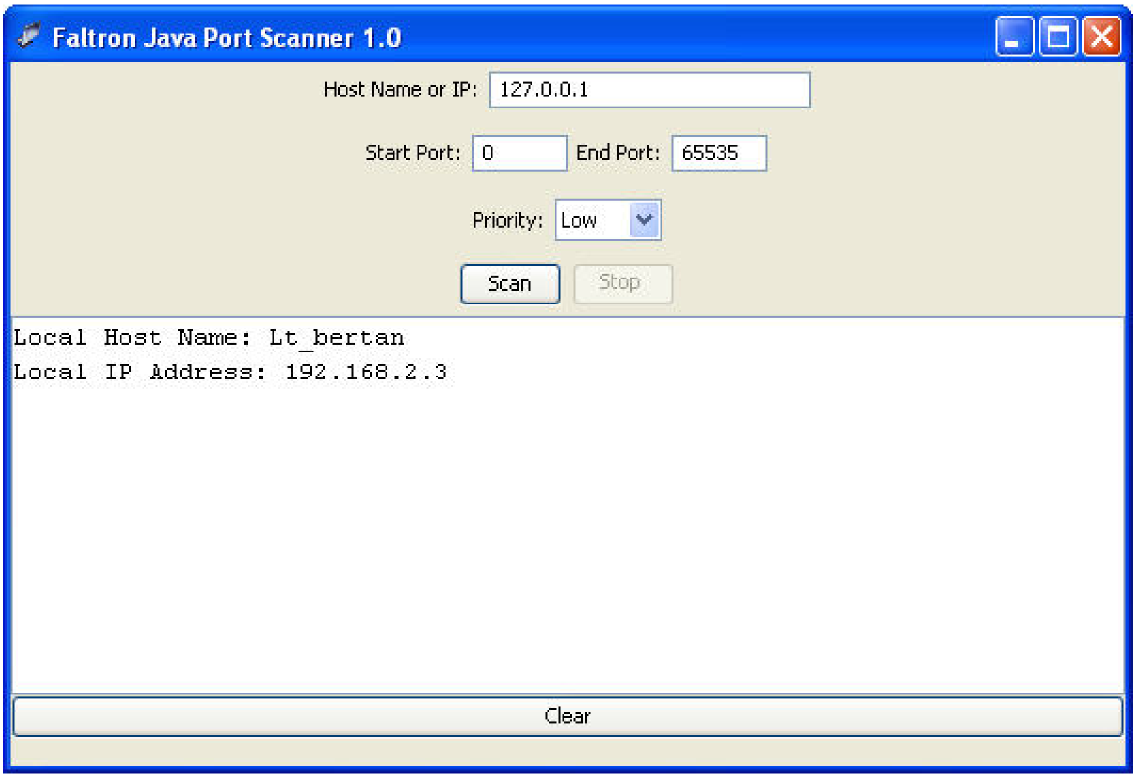
☑ TCP ☐ UDP

☑ Port Range

1 to 1024

```
GNU Java Network Port Scanner - JMap


Hostname:      IP address or valid name lookup
Subnet:        Range of addresses/hosts to scan
TCP/UDP:       Protocol to scan for, both may be selected
Port Range:     Enter port range to scan, otherwise defaults will be used

When scanning subnet, can select to scan for a specified service


Provided without warranty, and distributed under the terms of the GNU General
Public License

Written by Tom Salmon
```

Figure 5.2. JMAP Java Port Scanner.



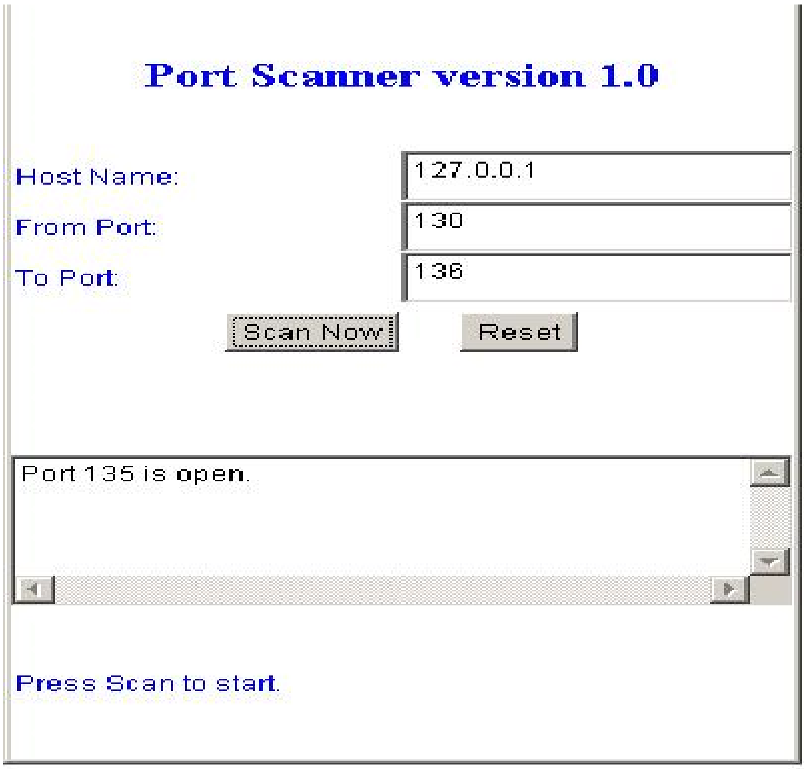
Host Name or IP: 127.0.0.1

Start Port: 0 End Port: 65535

Priority: Low

```
Local Host Name: Lt_bertan
Local IP Address: 192.168.2.3
```

Figure 5.3. Faltron Java Port Scanner.



Host Name: 127.0.0.1

From Port: 130

To Port: 136

```
Port 135 is open.
```

Press Scan to start.

Figure 5.4. A port scanner.

5.2.1. JMAP Java Port Scanner

JMAP is a Java network port scanner, a security tool to identify open ports on any host or network subnet. It features the ability to scan every host in a given network segment for a range of ports or a specific service. Both TCP and UDP are supported.

Boundary overflow vulnerability checker tool is evaluated for JMAP. Local variables, method/constructor parameters and input contracts are given in Table 5.1 and Table 5.2. Five input contracts have been used and one of them has been found as shown in Table 5.3.

Table 5.1. JMAP Local Variables and Method/Constructor Parameters.

Variables	Parameters
i (int)	port (int)
selectedPortNumber (int)	lowestPort (int)
partOne (int)	highestPort (int)
partTwo (int)	status (int)
partThree (int)	proto (int)
partFour (int)	
port (int)	
p (int)	

Table 5.2. JMAP Input Contracts.

No	Left Operand	Operator	Right Operand
1	lowestPort	>	0
2	lowestPort	<=	65535
3	highestPort	>	0
4	highestPort	<=	65535
5	lowestPort	<	highestPort

Table 5.3. JMAP Output.

Input Contract	
(lowestPort > 0)	HAS BEEN FOUND *File Name : null* *Class Name : Scan* *Method Name :* *Package Name :* *Line Number : 171*
(lowestPort <= 65535)	NOT FOUND
(highestPort > 0)	NOT FOUND
(highestPort <= 65535)	NOT FOUND
(lowestPort < highestPort)	NOT FOUND

5.2.2. Faltron Java Port Scanner

Faltron Java Port Scanner is an open source and very simple port scanner written in Java. It maximize the use of multi-threading by creating as much threads as possible, thus speeding up the scanning process.

Boundary overflow vulnerability checker tool is also evaluated for Faltron. Local variables, method/constructor parameters and input contracts are given in Table 5.4 and Table 5.5. Five input contracts have been used and none of them has been found as shown in Table 5.6.

Table 5.4. Faltron Local Variables and Method/Constructor Parameters.

Variables	Parameters
i (int)	port (int)
priority (int)	startPort (int)
appheight (int)	endPort (int)
appwidth (int)	

Table 5.5. Faltron Input Contracts.

No	Left Operand	Operator	Right Operand
1	startPort	>	0
2	startPort	<=	65535
3	endPort	>	0
4	endPort	<=	65535
5	startPort	<	endPort

Table 5.6. Faltron Output.

Input Contract	
(startPort > 0)	NOT FOUND
(startPort <= 65535)	NOT FOUND
(endPort > 0)	NOT FOUND
(endPort <= 65535)	NOT FOUND
(startPort < endPort)	NOT FOUND

5.2.3. A Java Port Scanner

Boundary overflow vulnerability checker tool is finally evaluated for "A Java Port Scanner". Local variables, method/constructor parameters and input contracts are given in Table 5.7 and Table 5.8. Five input contracts have been used and none of them has been found as shown in Table 5.9.

Table 5.7. A Java Port Scanner Local Variables and Method/Constructor Parameters.

Variables	Parameters
fp (int)	
tp (int)	
i (int)	

Table 5.8. A Java Port Scanner Input Contracts.

No	Left Operand	Operator	Right Operand
1	Fp	>	0
2	Fp	<=	65535
3	Tp	>	0
4	Tp	<=	65535
5	Fp	<	tp

Table 5.9. A Java Port Scanner Output.

Input Contract	
(fp > 0)	NOT FOUND
(fp <= 65535)	NOT FOUND
(tp > 0)	NOT FOUND
(tp <= 65535)	NOT FOUND
(fp < tp)	NOT FOUND

An overview of three test runs is given in Table 5.10. It is evident that only JMap has a control mechanism on startPort>0 and the other tools do not have any control mechanisms for the out of boundary input values causing boundary overflow.

Table 5.10. Comparison of the three test runs.

Software	Input Contracts				
	startPort >0	startPort <=65535	endPort >0	endPort <=65535	startPort < endPort
JMap	√	-	-	-	-
Faltron	-	-	-	-	-
A Java Port Scanner	-	-	-	-	-

Tuglular et al. (2009b) have developed a numerical input validation analysis tool in Java that enables a semi automatically detection of boundary overflow errors. The class of "Assertions" and the function of "Require" have been used for exception handling and numerical input validation, respectively. The developed tool has displayed the conditions of variables as well as whether or not condition checks exist in the source code related to numerical input relation. Three port scanners have been used to evaluate their approach that combines input validation with static analysis for evaluating given constraints. The user interface behaviour of port scan function has been modelled by using decision table augmented event sequence graphs. Test cases were generated for minimum and maximum port from the decision table and test of the port scan function has been evaluated in a real network environment. The boundary overflow related vulnerabilities have been detected and corrected by analyzing the source directory. The comparison has been also made between the faults detected before and after applying the boundary overflow detection algorithm.

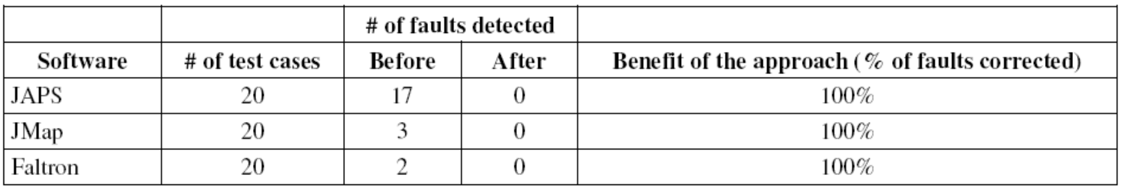
Netfender Firewall (version 1.5) was used to the evaluation and test pair has been generated based on equivalence class testing and boundary value approach. The algorithm has created a list of each of constraint containing the conditions of the variables and has generated the min and max test case pairs. They concluded that the cases with out of boundary input pairs cause problems in the network environment. It was analysed that faulty input pairs that are out of boundary values but the program behaves as they are not faulty because it does not abandon processing the related task. They also stated that the original software does not have control mechanism for the out of boundary input values causing boundary overflow.

Multiscan (version 0.8.5) and Pscan were also utilized which are again open source port scanners coded in C++ with scan a range of IP addresses and ports. They observed that these evaluations also have no exception handling mechanisms. They found that the control mechanisms against out of boundary values are deficient for the three port scanners. On the other hand, after the insertation of their control statements related to boundary constraints, the software has outputted the right error message and aborted sending the packets. Therefore, tool developed by Tuglular et al. (2009b) has been successfully implemented for detection and correction operations for finding deficiencies in the exception handling mechanism concerning boundary overflow problems in software development.

They also claimed that JMap has no error or exception raising mechanisms. The control mechanisms for pre- and post-conditions were deficient for this port scanner as well. They reported that Faltron has control mechanisms for pre-conditions but no mechanism has been found for post-conditions. It can be observed in Table 5.11

three port outputs considerably differ after the application of corrections into all three port scanners with respect to input contract.

Table 5.11. Comparison of the three port scanners
(Source: Tuglular et al., 2009b)



		# of faults detected		
Software	**# of test cases**	**Before**	**After**	**Benefit of the approach (% of faults corrected)**
JAPS	20	17	0	100%
JMap	20	3	0	100%
Faltron	20	2	0	100%

CHAPTER 6

CONCLUSIONS

In this work, we have made progress toward characterizing the software defects that can be found using static analysis techniques in programs written in Java. This study also dealt with the comparison of the output of different static analysis tools. Based on the literature review, it is obvious that PMD is a useful tool to apply to projects to analyze code and catch errors which otherwise can be missed with code reviews and inspections. Since it is flexible allowing customizations, it is advantageous to apply specific rules to projects that make PMD a valuable tool to use. A new tool that uses PMD has been implemented to find boundary overflow vulnerability. The overflow vulnerability checker tool has successfully carried out detection operations in the source code related to overflow vulnerabilities. It is observed that the boundary overflow vulnerabilities are not considered and neglected throughout the software development. Therefore, the overflow vulnerability checker tool that is introduced in this thesis might contribute to prevent the undesirable situation that may occur as a result of the deficiencies in the software related to overflow vulnerabilities. For future work, extension of the approach is planned by considering input labels as elements of contracts and evaluating these contracts.

REFERENCES

Aldrich J.; Kostadinov V.; Chambers C. Alias Annotations for Program Understanding. *ACM SIGPLAN Notices* **2002,** 37 (11), 311-330.

Artzi S.; Kiezun A.; Glasser D.; Ernst M.D. Combined Static and Dynamic Mutability Analysis, *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, November 5–9, 2007, Atlanta, Georgia, USA.

Ball T.; Rajamani S.K. The SLAM Project: Debugging System Software via Static Analysis, *Proceedings of the 29th ACM SIGPLAN*, 2002, Oregon.

Balakrishnan, G., R.; Gruian, T. R.; Teitelbaum T. CodeSurfer/x86 – A Platform for Analyzing x86 Executables. International Conference on Compiler Construction. 2005.

Banerjee A.; Naumann D.A., A Static Analysis for Instance-based Confinement in Java. *ACM SIGPLAN Notices*, **2002**.

Barbuti R.; Giacobazzi R.; Levi G. A General Framework for Semantics-Based Bottom-up Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems* **1993,** 15(1), 133-181.

Clark D.; Drossopoulou S.; Müller P.; Noble J.; Wrigstad T. Aliasing, Confinement and Ownership in Object-Oriented Programming, *ECOOP 2008*, Workshops, Paphos, Cyprus.

Clarke E.M.; Jha S.; Lu Y.; Weith H. Tree-Like Counterexamples in Model Checking. *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science* (LICS'02), 2002.

Codehous.org
http://jaxen.codehaus.org/ (accessed April 4, 2010).

Copeland T. PMD Applied**,** Centennial Books, 2005.

Cousot, P.; Cousot R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *ACM Symposium on Principles of Programming Languages*. 1977, Los Angeles, CA.

Cousot P.; Cousot R. Refining Model Checking by Abstract Interpretation. *Automated Software Engineering* **1999,** 6, 69–95.

Cowan, C. Software Security for Open-Source Systems. *IEEE Security and Privacy Magazine* **2003,** 1(1), 38-45.

Crew R.F., ASTLOG: A Language for Examining Abstract Syntax Trees. *Proceedings of the Conference on Domain-Specific Languages,* October 1997, Santa Barbara, California, USA.

Denning, D. E. A Lattice Model of Secure Information Flow. *Communications of the ACM* **1976**, 19(5), 236-243.

England M. Paper Evaluation: Securing Web Application Code by Static Analysis and Runtime Protection. **2008.**

Emanuelsson P.; Nilsson U. *A Comparative Study of Industrial Static Analysis Tools*, Technical reports in Computer and Information Science, Department of Computer and Information Science, Linkoping University, Sweden, 2008.

FaltronSoft.org http://faltronsoft.org/index.php?option=com_content&task=view&id=23&Itemid=29) (accessed April 4, 2010).

Foster, J. S.; Fähndrich, M.; Aiken, A. A Theory of Type Qualifiers. *Proceeding of ACM SIGPLAN 1999 Conf. Programming Language Design and Implementation* (PLDI'99), pp 192-203, 34(5) of ACM SIGPLAN Notices, Atlanta, Georgia, May 1- 4, 1999.

Hsu A.., Jagannathan S.; Mustehsan S., Mwamufiya S.; Novakouski M. *Analysis Tool Evaluation:PMD*, Final Report, School of Computer Science Carnegie Mellon University, 2007.

Hovemeyer D.; Pugh W. Finding Bugs is Easy. *ACM SIGPLAN Notices*, **2004** 39 (12), 92-106.

Huang Y-W; Yu F.; Hang C.; Tsai C-H. ; Lee D-T. ; Kuo S-Y. Securing Web Application Code by Static Analysis and Runtime Protection. *Proceedings of the 13th International Conference on World Wide Web,* May 17–22, 2004, New York, New York, USA.

McCabe T.J. A Complexity Measure. *IEEE Transactions on Software Engineering,* **1976,** 2(4), 308-320.

Mc Leen, J. A Comment on the "Basic Security Theorem" of Bell and LaPadula *Information Processing Letters* **1985,** 20, 67-70.

Mc Leen, J., Information Flow Is a Way of Modelling Data Flow,1990.

Mell P.; Tracy M.C. Procedures for Handling Security Patches, NIST Special Publication, 800-40, 2002.

Morrisett, G.; Walker, D.; Crary, K.; Glew, N. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, **1999**, 21(3), 528-569.

Muftuoglu C.A., “A Detection and Correction Approach for Overflow Vulnerabilities in Graphical User Interfaces”, M.S. Thesis, İzmir Institute of Technology, İzmir, 2009.

Myers, A. C. JFlow: Practical Mostly-Static Information Flow Control. *Proceedings of 26th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL’99)*, p. 228-241, 1999, San Antonio, Texas.

Necula, G. C. Proof-Carrying Code. *Conference Record of the 24th Annual ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL’97),* p. 106-119, Jan 1997, Paris, France.

OpenSource.org
http://opensource.org (accessed April 4, 2010).

Orbaek, P. Can You Trust Your Data? *Proceddings of 1995 TAPSOFT/FASE Conference*, p.575-590, LNCS 915, May 1995, Aarhus, Denmark, Springer-Verlag.

Petersen P.M.; Padua D.A., Static and Dynamic Evaluation of Data Dependence Analysis. *Proceedings of the 7th International Conference on Supercomputing*, 1993, Tokyo, Japan.

Planet-Sourcecode.com
http://www.planet-sourcecode.com/vb/scripts/ShowCode.asp?txtCodeId=2735&lngWId=2 (accessed April 4, 2010).

Pottier, F.; Simonet, V. Information Flow Inference for ML. *ACM Transactions on Programming Languages and Systems*, **2003**, 25(1), 117-158,

Pugh W., A Practical Algorithm for Exact Array Dependence Analysis, *Communications of ACM,* **1992,** 35 (8), 102-114.

Rutar N.; Almazan C.B.; Foster J.S. A Comparison of Bug Finding Tools for Java, *Symposium on Software Reliability Eng. (ISSRE'04*), 2004, France.

Sabelfeld A.; Andrew C. M. Language-Based Information-Flow Security IEEE *Journal on Selected Areas in Communications,* **2003,** 21 (1).

Security Innovation.com
http://www.securityinnovation.com/security-report/november/staticAnalysis1.htm (accessed April 4, 2010).

Shepperd M., A Critique of Cyclomatic Complexity as a Software Metric, *Software Engineering Journal*, March **1988,** 30-36.

Software Technology Support Center
http://www.stsc.hill.af.mil/crossTalk/2008/06/0806Anderson.html (accessed April 4, 2010).

SourceForge.net (a)
http://pmd.sourceforge.net (accessed April 4, 2010).

SourceForge.net (b)
http://checkstyle.sourceforge.net (accessed April 4, 2010).

SourceForge.net (c)
http://jlint.sourceforge.net (accessed April 4, 2010).

SourceForge.net (d)
http://qjpro.sourceforge.net (accessed April 4, 2010).

Stamelos I.; Angelis L.; Oikonomou A.; Bleris G.L. Code Quality Analysis in Open Source Software Development, *Info Systems J.* **2002,** 12, 43–60.

TomSalmon.com
http://tomsalmon.com/jmap.php (accessed April 4, 2010).

Tuglular T.; Müftüoğlu C.A.; Kaya O.; Belli F.; Linschulte M. GUI-Based Testing of Boundary Overflow Vulnerability, 33rd Annual IEEE International Computer Software and Applications Conference, 2009a, Seattle, Washington, USA.

Tuglular T., Muftuoglu C. A., F. Belli, Linschulte M., "Event-Based Input Validation Using Design-by-Contract Patterns", 20th Annual International Symposium on Software Reliability Engineering (ISSRE 2009), 2009b, Mysuru, India.

Volpano D.; Smith G.; Irvine C. A Sound Type System for Secure Flow Analysis, CS 711: Language-Based Security and Information Flow, September 2003.

Wagner S.; Jurjens J.; Koller C.; Trischberger P. Comparing Bug Finding Tools with Reviews and Tests**,** *TestCom 2005,* **2005**, LNCS 3502, 40–55.

Ware M.S.; Fox C.J. Securing Java Code: Heuristics and an Evaluation of Static Analysis Tools *SAW '08* June 12, 2008, Tucson, Arizona, USA.

APPENDIX A

OPEN SOURCE TOOLS

A.1. Multi-language

- RATS — Rough Auditing Tool for Security, which can scan C, C++, Perl, PHP and Python source code.
- YASCA — Yet Another Source Code Analyzer, a plugin-based framework for scanning arbitrary file types, with plugins for scanning C/C++, Java, JavaScript, ASP, PHP, HTML/CSS, ColdFusion, COBOL, and other file types. It integrates with other scanners, including FindBugs, JLint, PMD, and Pixy.
- CPD — The Copy/Paste Detector (CPD) is an add-on to PMD tat finds duplicated code. CPD works with Java, JSP, C, C++, Fortran and PHP code.

A.2. . NET (C#, VB.NET and all .NET Compatible Languages)

- FxCop — Free static analysis for Microsoft .NET programs that compile to CIL. Standalone and integrated in some Microsoft Visual Studio editions. From Microsoft.
- StyleCop — Analyzes C# source code to enforce a set of style and consistency rules. It can be run from inside of Microsoft Visual Studio or integrated into an MSBuild project. Free download from Microsoft.

A.3. Java

- Checkstyle — Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. It automates the process of checking Java code to spare humans of this boring (but important) task. This makes it ideal for projects that want to enforce a coding standard. This tool that analyzes source code to find layout issues, class design problems, duplicate code and bugs. Besides some static code analysis, it can be used to show violations of a configured coding standard (sourceforge.net (b), 2010).

- FindBugs — an open-source static bytecode analyzer for Java (based on Jakarta BCEL) from to find occurrences of “bug patterns”, which are code idioms that are likely to be errors (Hovemeyer and Pugh 2004). The tool *FindBugs* was developed at the University of Maryland and can detect potentially problematic code fragments by using a list of bug patterns. It can find faults such as dereferencing null-pointers or unused variables. To some extent, it also uses dataflow analysis for this.
- PMD — a static rule set based Java source code analyzer that identifies potential problems. This tool concentrates on the source code and is therefore especially suitable to enforce coding standards. It finds, for example, empty try/catch blocks, overly complex expressions, and classes with high cyclomatic complexity. It can be customised by using XPath expressions on the parser tree. This tool that scans source code to detect potential bugs, dead code, suboptimal code, overcomplicate expressions, and duplicate code (sourceforge.net (a), 2010). In addition to some detection of clearly erroneous code, many of the “bugs” PMD looks for are stylistic conventions whose violation might be suspicious under some circumstances. For example, having a try statement with an empty catch block might indicate that the caught error is incorrectly discarded. Because PMD includes many detectors for bugs that depend on programming style, PMD includes support for selecting which detectors or groups of detectors should be run (Rutar et al., 2004).
- Jlint — a tool that analyzes Java source code and bytecode to detect bugs, inconsistencies, and problems with synchronization by performing “data flow analysis and building the lock graph. JLint also includes an interprocedural, inter-file component to find deadlocks by building a lock graph and ensuring that there are never any cycles in the graph (sourceforge.net (c), 2010).
- QJ Pro. — a tool that analyses the source code. It supports over 200 rules including ignored return values, too long variable names, or a disproportion between code and commentary lines. It is also possible to define additional rules. Furthermore, checks based on code metrics can be used. The possibility to use various filters is especially helpful in this tool (sourceforge.net (d), 2010).

- Hammurapi — (Free for non-commercial use only) versatile code review solution. Hammurapi is an open source code inspection tool. Its release comes with more than 100 inspectors which inspect different aspects of code: Compliance with EJB specification, threading issues, coding standards.
- Sonar — a platform to manage source code quality. Sonar is a continuous quality control tool for Java applications. Its basic purpose is to join your existing continuous integration tools to place all your development projects under quality control.
- Soot — a language manipulation and optimization framework consisting of intermediate languages for Java
- Squale — a platform to manage software quality (also available for other languages, using commercial analysis tools though)

A.4. C

- BLAST (Berkeley Lazy Abstraction Software verification Tool) — a software model checker for C programs based on lazy abstraction.
- Clang — A compiler that includes a static analyzer.
- Frama-C — A static analysis framework for C.
- Sparse — A tool designed to find faults in the Linux kernel.
- Splint — An open source evolved version of Lint (C language).
- Uno — A tool designed to find most common type of programming errors without generating too much output.

A.5. C++

- Cppcheck — can find memory leaks, buffer overruns and many other common errors.
- compass - project of rose compiler framework.

A.6. Objective-C

- Clang — the free Clang project includes a static analyzer. As of version 3.2, this analyzer is included in Xcode.

A.7. Perl

- Perl::Critic — module and program to help find deviations from commonly accepted best practices

A.8. ActionScript

- Apparat — a language manipulation and optimization framework consisting of intermediate representations for ActionScript.
- AS3V — a static ruleset based analyzer focussing on performance leaks.
- FlexPMD — a static ruleset based ActionScript source code analyzer that identifies potential problems; based on PMD.

APPENDIX B

COMMERCIAL TOOLS

B.1. Multi-language

- Axivion Bauhaus Suite — a tool for C, C++, C#, Java and Ada code that comprises various analyses such as architecture checking, interface analyses, and clone detection.
- Checkmarx - a tool to identify, track and fix technical and logical security flaws from the root: the source code. Analyzes .Net, Java, Classic ASP, C/C++ and Salesforce.com's Apex and Visual Force.
- CodeSecure — Appliance with Web interface and built-in language parsers for analyzing ASP.NET, VB.NET, C#, Java/J2EE, JSP, EJB, PHP, Classic ASP and VBScript.
- CAST Application Intelligence Platform — Detailed, audience-specific dashboards to measure quality and productivity. 30+ languages, SAP, Oracle, PeopleSoft, .NET, Java, C/C++, Struts, and all major databases.
- CodeScan Labs CodeScan Developer — identifies security vulnerabilities and issues in ASP classic, PHP, ASP.Net, C#.Net source code
- Coverity Prevent — identifies security vulnerabilities and code defects in C, C++, C# and Java code.
- DMS Software Reengineering Toolkit — supports custom analysis of C, C++, Java, COBOL, and many other languages.
- Compuware DevEnterprise — analysis of COBOL, PL/I, JCL, CICS, DB2, IMS and others.
- Fortify — helps developers identify software security vulnerabilities in C/C++, .NET, Java, JSP, ASP.NET, ColdFusion, "Classic" ASP, PHP, VB6, VBScript, JavaScript, PL/SQL, T-SQL and COBOL as well as configuration files.
- GrammaTech CodeSonar — Analyzes C,C++. Ada-Assured -Analyzes Ada

- Klocwork Insight and Klocwork Developer for Java — provides security vulnerability and defect detection as well as architectural and build-over-build trend analysis for C, C++, C# and Java
- Lattix, Inc. LDM — Architecture and dependency analysis tool for Ada, C/C++, Java, .NET software systems.
- LDRA Testbed — A software analysis and testing tool suite for C, C++, Ada83, Ada95 and Assembler (Intel, Freescale, Texas Instruments).
- Ounce Labs — automated source code analysis that enables organizations to identify and eliminate software security vulnerabilities in languages including Java, JSP, C/C++, C#, ASP.NET, and VB.Net.
- Parasoft — Security, reliability, performance, and maintainability analysis of Java, JSP, C, C++, .NET (C#, ASP.NET, VB.Net, etc.), WSDL, XML, HTML, CSS, JavaScript, VBScript/ASP, and configuration files.
- SofCheck Inspector — provides static detection of logic errors, race conditions, and redundant code for Java and Ada.
- Sotoarc/Sotograph — Architecture and quality in-depth analysis and monitoring for Java, C#, C and C++
- Structure101 — For understanding, analyzing, measuring and controlling the quality of Software Architecture as it evolves over time. Available for Java and Ada, with support for C/C++ via Coverity and Programming Research.
- Understand — analyzes C,C++, Java, Ada, Fortran, Jovial, Delphi — reverse engineering of source, code navigation, and metrics tool.
- Visual Studio Team System — analyzes C++,C# source codes. only available in team suite and development edition.

B.2. .NET

- ReSharper — Add-on for Visual Studio 2003/2005 from the creators of IntelliJ IDEA, which also provides static code analysis for C#.
- NDepend — Simplifies managing a complex .NET code base by analyzing code dependencies, by defining design rules, by doing impact analysis, and by comparing different versions of the code (all .NET languages supported).

- CodeIt.Right — combines Static Code Analysis and automatic Refactoring to best practices which allows automatically correct code errors and violations. Supports both C# and VB.NET.
- Gendarme — extensible rule-based tool to find problems in .NET applications and libraries, particularly those that contain code in ECMA CIL format.

B.3. C/C++

- Abraxas Software CodeCheck — programmable static analysis and style checker for C and C++ code.
- Astrée — Run-time error analyzer for C
- Green Hills Software DoubleCheck — static analysis for C and C++ code.
- HP Code Advisor — A static analysis tool for C and C++ programs
- LDRA Testbed — A software analysis and testing tool suite for C & C++.
- Microsoft PREfast — The "Analyze Tool" included with Microsoft Visual Studio Team Editions.
- Microsoft PREfast for Drivers (PFD) — An extension to PREfast to allow better analysis of Windows device drivers.
- Microsoft Static Driver Verifier (SDV) — Performs detailed code path analysis for Windows device drivers.
- PAG — The Program Analyzer Generator.
- PC-Lint — A software analysis tool for C & C++.
- QA-C (and QA-C++) — deep static analysis of C for quality assurance and guideline enforcement.
- Red Lizard's Goanna — Static analysis for C/C++ in Eclipse and Visual Studio.
- Viva64 — analyzes C, C++ code to detect 64-bit portability issues.
- CppDepend — Simplifies managing a complex C++ code base by analyzing code dependencies, by defining design rules, by doing impact analysis, and by comparing different versions of the code.

B.4. Java

- checKing — monitors the quality of software development process, including violations of coding rules for Java, JSP, Javascript, XML and HTML.
- IntelliJ IDEA — IDE for Java that also provides static code analysis.
- Swat4j — a model based, goal oriented source code auditing tool for Java.

B.5. Visual Basic

- Project Analyzer — static analysis tool for Visual Basic, Visual Basic .NET and Visual Basic for Applications.

APPENDIX C

RULESETS OF PMD

C.1. Basic (rulesets/basic.xml)

These are basic rules which need to be followed. Across BIRT, Nomad PIM, jLibrary and PaperDog the following conditions were observe:

1. Catch blocks shouldn't be empty,
2. Override hashCode() anytime equals() is overridden
3. Nested if statements should be avoided
4. If statements evaluating to only true or only false should be avoided
5. Avoid temporary variables when converting from primitive datatype to String
6. Empty statements in loops should be avoided
7. Avoid instantiating data type values which have existing constants defined for ex. instantiating Boolean type with constructor, instead of using Boolean.true, instantiating BigDecimal (0) instead of using BigDecimal.ZERO.
8. Final modifier in a final class is redundant
9. Overridden methods contain only call to super() and nothing else.
10. Avoid unnecessary return statements
11. Do not start a literal by 0 unless it's an octal value

There are some false positives in the above cases. In many cases, catch blocks were known to the developer and contained comments explaining why the exception was applicable or not. For final modifiers, the code had final modifiers for all the methods inside the final class and each line was shown as a warning. This though can be avoided is not an error. Some methods in classes were overridden but did not have any special task so it just had a call to super. This was flagged as error in the PMD results but this again does not cause erroneous behaviour. 0 was used to denote octal values, so that also turned out to be a false positive. Overall this ruleset is useful and because it catches basic errors which will be missed during reviews, inspection or testing.

C.2. Naming (rulesets/naming.xml)

This ruleset tests for the standard Java naming conventions. Some typical errors which were observed during this analysis were:

1. Abstract classes should have the name AbstractXXX
2. Variable names should not be too short
3. Field names matching class names lead to confusion
4. Variable names and method names should not be too long or too short
5. Variable names which are not constant should not contain "_" (underscore)
6. Class names should begin with an uppercase letter, method and field names should
7. Begin with a lowercase letter
8. The field name indicates a constant but its modifiers do not

The code is strictly checked against Java naming conventions. This resulted in a lot of false positives. The results for all the four projects were filled with the entries about variable names being too short or method names being too short or too long. The one error message which said "The field name indicates a constant but its modifiers do not" sounded to be a valid error situation but on analyzing the code it was found that a variable name was completely in uppercase and had a underscore character which according to Java naming convention is to be used only for constants. Hence it was not an error in code but an error in naming.

C.3. Unused Code (rulesets/unusedcode.xml)

This ruleset checks for unused code in the project. The following are typical error messages which occurred for the projects considered:

1. Private fields and local variables that are never read,
2. Private methods that are never called,
3. Unused method parameters and constructor parameters,
4. Unreachable statements.

With the widespread use of IDEs for Java like Eclipse and JBuilder, this ruleset is not very useful. For example in Eclipse, unused fields, variables and private methods are automatically highlighted with warning messages. Unused method parameters and constructor parameters are not enabled by default in Eclipse but settings can be changed to ensure that this check is included. Unreachable statements are treated as errors in Eclipse and the editor will not allow compilation till the error has been fixed. Hence this ruleset is not necessary.

C.4. Design (rulesets/design.xml)

This ruleset covered various good design principles. The errors observed in this case were many. Few important ones are listed below:

1. Switch statements should have default blocks,
2. If conditions of the form a!=b should be avoided
3. Private fields which are initialized only in the constructor and are not modified anywhere else should be made constant (final)
4. Deeply nested if blocks should be avoided
5. Parameters should not be reassigned
6. Replace calls like size() == 0 with isEmpty()
7. Overridable methods should not be called in the constructor
8. If all methods in a class are static then the class can be converted to Singleton
9. Caught exceptions should not be rethrown as the stack trace may be lost.
10. Unnecessary comparisons in Boolean expressions should be avoided
11. Equals() should be used for object comparison
12. It is better to use block level synchronization than method level synchronization
13. Literals should be used first in string comparisons
14. Values can be returned directly instead of assigning them to temporary variables and
15. Resources like connections should be closed after use

This rulesets identifies many conditions which need to be cleared in the code. This cannot be covered exhaustively using manual reviews or inspection. There is one suggestion for conversion to Singleton when static methods are used, this may or may not be used based on the architecture. Other than this, all conditions highlighted by the ruleset needs to be corrected. Design rulesets based on the data gathered on all the projects do not have instances of false positives. This ruleset is useful for analysis.

C.5. Import Statements (rulesets/imports.xml)

This ruleset checks for minor issues with import statements. The following were the three conditions observed across all packages:

1. Avoid duplicate imports
2. There is no need to import a class which resides in the same package
3. Avoid unused imports

Using IDEs makes this ruleset redundant. Eclipse has an option to reorganize imports. This automatically corrects imports, removes duplicate and unused imports, removes imports of the type java.io.* and includes individual classes. Hence this ruleset need not be used.

C.6. JUnit Tests (rulesets/junit.xml)

This ruleset looks for specific issues in the test cases and test methods. The issues observed were:

1. Assertions should have a message
2. Correct spelling of method names, especially JUnit keywords like setUp() and tearDown()
3. JUnit tests should contain an assert or fail
4. Classes which contain JUnit test cases should end with Test
5. Use assertSame(x, y) instead of assertTrue(x==y)

There were two main problems observed with this ruleset. This ruleset should be run only on the JUnit source files. If it is run on the main project, then each java file is considered as a JUnit test and the report contains a higher percentage of false positives. Especially the messages “Assertions should have a message” and “Incorrect method names setUp() and tearDown()” occur when method names setup() is used in a normal class file or assert statement is used within non-JUnit code. There is another false positive while running the code on JUnit test cases. If a test method calls another method which does the assert() or fail() and does not have assert or fail in the method body then the ruleset is not able to recognize it and generates errors. The ruleset does not verify if the test case is valid or not. It checks for syntax errors. Hence it does not add any value. Code review for test cases should help in verifying errors. This ruleset need not be used.

C.7. Strings (rulesets/string.xml)

This ruleset identifies problems that occur while using String and StringBuffer. The common errors observed are:

1. Avoid duplicating string literals
2. Appending characters to StringBuffer should be avoided
3. Calling String.valueOf() to append to a string is not necessary
4. String.trim().length() == 0 is inefficient to check if string is empty
5. Constructor for StringBuffer is initialized with a smaller number and more characters
6. Are appended (buffer overflow)
7. Using equalsIgnoreCase() is efficient instead of converting strings to upper or lower
8. Case and then comparing.
9. IndexOf(char) is faster then indexOf(String)
10. Calling the String constructor, and calling toString() on String objects is unnecessary

This ruleset covers various conditions for String and StringBuffer usage. Since String objects are used heavily, it is beneficial to run this test case and correct the related errors. Almost all errors are relevant and there were no false positives observed. This helps in avoiding buffer overflows and improving performance and memory usage by proper allocation of strings and calling appropriate methods.

C.8. Braces (rulesets/braces.xml)

This ruleset checks for, if, while, and else statements and the usage of braces. All the results were of the same type, which is

1. Avoid using if and else without curly braces

IDEs like Eclipse create templates for if-else-elseif statements which include the curly braces using the auto complete option (this is not a default option and needs to be enabled). Hence this ruleset need not be used.

C.9. Javabeans (rulesets/javabeans.xml)

This ruleset inspects JavaBeans components. The typical errors observed are:

1. Non-transient and non-static members need to be marked as transient or accessors should be provided
2. Classes which implement Serializable should have a serialVersionUID

The first error indicates non compliance with JavaBean coding standard. This can be caught during reviews. The second error is usually shown as a warning in Eclipse IDE. This ruleset can be used if JavaBeans is used for coding. If not this does not add any value.

C.10. Finalizers

This ruleset identifies two types of errors overall:

1. If finalize is used it should be protected
2. Last call in finalize should be a call to super.finalize

Since finalize is being used very rarely in current implementations it is not necessary to use this ruleset. For the few situations where it is used, it is easy to remember the two conditions and can be checked during reviews.

C.11. Clone (rulesets/clone.xml)

There are only a few rules for clone() methods. They are:

1. Classes that override clone() must implement Cloneable,
2. Clone() methods should call super.clone()
3. Clone() methods should be declared to throw

The main problem with clone() method is in the case of deep copy and shallow copy. This ruleset does not verify if that is achieved. Since there are only three rules, it is fair to assume that it should be included as a part of coding standard and should be an item in the review checklist. This ruleset can be run once at the end to verify if the standard is met and need not be run always.

C.12. Coupling (rulesets/coupling.xml)

The errors which are generated with this ruleset are:

1. Too many imports or too many different objects indicate coupling
2. Avoid usage of subclass types like Vector, ArrayList and HashMap and use the supertype or interface instead

There are false positives when this ruleset is run. In projects which use various other projects for functionality like using Tomcat, log4j, JBoss, Hibernate, Eclipse RCP, Struts 2 and BIRT which are typical for any web applications, there are usually too many imports and too many different objects. Since the report is full of such warnings it is difficult to find the useful messages. Usage of subclasses again should be a coding convention and should be included in code reviews. This ruleset does not add value.

C.13. Strict Exceptions (rulesets/strictexception.xml)

The errors generated with this ruleset are:

1. Raw exception types should not be thrown
2. Methods should not be declared to throw java.lang.Exception,
3. Avoid throwing null pointer exceptions
4. Catch should not throw the exception caught (this is also found in design ruleset)
5. Throwable should not be caught

This ruleset is helpful because exception handling is something which needs to be done well, else the code may crash due to unforeseen errors. Since PMD generates exhaustive analysis it is easy to not miss out conditions. The analysis results observed did not contain any false positives.

C.14. Controversial (rulesets/controversial.xml)

This ruleset has some conditions which cannot be followed in practice. The typically observed warnings are as follows:

1. Each class should have at least one constructor
2. A method should have only one exit point
3. Avoid unnecessary constructors
4. It is good practice to call super() in constructors
5. Captures data flow anomalies
6. Use explicit scoping rather than default package private scope
7. Don’t assign null to an object

It is not possible to always have one exit point. It is often considered that assigning an object to null as initial state is a good practice. Some of PMD's rules are valid and some are arguable hence the name controversial ruleset. This does not add value to the code analysis because it does not catch errors which might break the system or might cause the system to be insecure. Hence the ruleset need not be used.

C.15. Logging (rulesets/logging-java.xml, rulesets/ logging-jakarta-commons.xml)

This ruleset checks for usage of logging. The errors identified are as follows:

1. Logger variables should be static and final
2. System.out.print and println, and printStackTrace should be replaced with calls to logging

This ruleset is not very helpful. There are too many false positives which are generated. In most cases System.out.println and printStackTrace are used reliably. Moreover print messages are easily observable during unit testing or integration testing and hence can be corrected.

C.16. J2EE (rulesets/j2ee.xml)

This ruleset checks for compliance with J2EE architecture. Since none of the projects used for this report followed a J2EE architecture there was only one error which was produced throughout which was usage of getClassLoader(). This ruleset is not applicable for non-J2EE projects.

C.17. Optimizations (rulesets/optimizations.xml)

This ruleset covers certain optimization conditions. The typical conditions are:

1. Parameters, fields or variables not assigned should be declared as final
2. ArrayList can be used instead of Vector
3. Use StringBuffer instead of += for concatenating strings

C.18. Type Resolution (rulesets/typeresolution.xml)

This ruleset captured only two types of error conditions

1. Classes that override clone() must implement Cloneable,
2. Avoid usage of subclass types like Vector, ArrayList and HashMap and use the supertype or interface instead

This ruleset covers conditions which are already covered in Cloning and Coupling rulesets. These conditions do not add specific value and they can be covered as a part of coding standards and review checklists.

C.19. Unsecure Code (rulesets/sunsecure.xml)

This ruleset checks for array assignments. In particular it generates the following conditions:

1. Internal arrays being stored directly
2. Return variables which expose internal arrays

These are conditions which need to be checked to ensure that arrays are handled correctly. Hence this ruleset needs to be executed.