Carnegie Mellon University

Prevent Vs FindBugs Application and Evaluation

By
Lyle Holsinger
Snehal Fulzele
Smita Ramteke
Ken Tamagawa
Sahawut Wesaratchakit

For
Dr. Jonathan Aldrich
17654 Analysis of Software Artifacts
March 29, 2008

Contents

1. Introduction

Static analysis tools allow software developers to find defects before the software which helps improve their code quality. However, these tools also report a number of false positives and true positives on which the developers have to spend considerable amount of time. Additionally these tools can take a long time to analyze and thus they may not be widely adopted.

The focus of the report is the assessment of two static analysis tools viz. *Coverity Prevent* and *FindBugs*. Each tool was applied to the open-source JAVA project called *JEdit*. We briefly introduce the tools and present the results of its application to JEdit. We then highlight their strengths and weaknesses in terms of the *false positives* and *true positives* they report. We also highlight the differences between the two tools with respect to the analysis technique used, category of defects found and the analysis time taken. Lastly, we present our conclusion and recommendation.

2. Coverity Prevent

Overview

Prevent is a static analysis tool distributed by Coverity that locates defects, security vulnerabilities, and concurrency issues in C/C++ and Java code. The technology that drives Prevent's analyses was developed in the Computer Systems Laboratory at Standford University in Palo Alto, CA from 1998 to 2002. The practical industry application of the research efforts by the Standford team were quickly recognized and members of this team soon developed the tool into a commercial product.

Prevent is now used widely in development of software. Coverity's customers come from a diverse set of industries. Some high profile users of Prevent include Samsung, UBS, Westinghouse, Mozilla, the FDA, Phillips, LG, and Cisco. In a contract with the U.S. Department of Homeland Security, Coverity's technology is being used to identify defects and security vulnerabilities in open source projects. Since March 2006, it has identified and helped to fix over 7500 defects across 250 open source projects and 50 million lines of code.

Separate licenses are available for C/C++ analysis and Java analysis. Coverity does not list their price publicly. However, one article [1] claims that the cost of a one year license is based on the number of lines of code, regardless of the number of users. The cost listed in this 2006 article is \$50,000 for 500,000 lines of code.

Coverity Prevent uses a combination of analysis techniques to accurately and efficiently identify defects. Interprocedural dataflow analysis traces values through a program's call and return paths in order to locate defects, such as null pointer dereferences, that are extremely difficult to identify through manual inspection. False-path pruning is used to improve the efficiency and speed of analyses by removing from the consideration of defect checkers paths that will not generate defects or are infeasible at runtime. Statistical analysis is performed during code analysis in order to recognize anomalies and inconsistencies in results. Through incremental analysis, prevent reduces analysis time by precisely tracking and storing compilation data so that only changed or affected code is analyzed during subsequent builds. Finally, a powerful Boolean Satisfiability (SAT) engine is used in defect checking and false-path pruning. This engine essentially considers different TRUE/FALSE assignments to variables in order to arrive at facts about the code.

Prevent is capable of identifying a wide variety of defects. Some of the categories include:

- **Concurrency problems** including deadlocks and misuse of call blocking;
- **Performance degradation** including memory leaks, file handle leaks, custom memory and network resource leaks, and database connection leaks;
- **Crash-causing defects** including null pointer dereferences, use-after-free defects, double-free defects, improper memory allocation, and mismatched array new/delete;
- **Incorrect Program behavior** including dead code caused by logical errors, uninitialized variables, and invalid use of negative values;
- **Improper use of API's** including misuse of STL's and API error handling.

Experiment Setup

Prevent allows enabling and disabling of specific bug checkers. Furthermore, each check can be fine-tuned from the command prompt. In order to support a reasonable mapping between defects discovered by both Coverity Prevent and Findbugs, the majority of default defect checkers were enabled during analysis. JDBC Connection defect checking was disabled because it was irrelevant to our analysis of JEdit.

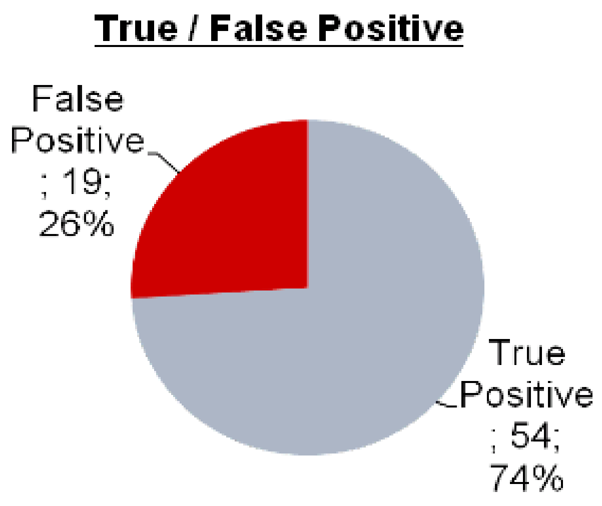
Experiment Results

In total 73 defects were identified by Prevent. The nature of these defects varied. 15 defects were found by the BAD_EQ checker, which flags comparisons that Prevent determines should use an identity comparison or an equality comparison. 8 defects were found by the CALL_SUPER checker, which flags missing calls to a superclass method in an override. 1 defect was found by the CLASS_CAST checker, which flags downcasting of an object when the value is extracted. 11 defects were found by the FORWARD_NULL checker, which flags dereferences that occur after a check against null. 27 defects were found by the NULL_RETURN checker, which flags dereferences that occur, unchecked, after a potential null return. 11 defects were found by the RESOURCE_LEAKS checker, which flags when resources are not released quickly enough for garbage collection.

Evaluation of these results was time consuming. To effectively evaluate the defects, it took approximately 5 person-hours to determine the validity of all 73 defects. This is because the defects Prevent uncovered were deeply interprocedural and required a thorough understanding of factors, both local and external, surrounding the defective line of code. Of the 73 defects, 54 were determined to be true positives. 19 were determined to be false positives. This number is inflated by the heuristic that Prevent uses to identify the correctness of object comparisons. The decision is made democratically based on all comparisons of a particular type of object. If the majority of comparisons are made using object identity (==), the remaining comparisons that use equality (.equals()) are considered to be defects. The converse is also true. As a result, defects identified by the BAD_EQ checker must be scrutinized carefully to determine if a defect truly exists or if the comparison is being performed in an appropriate context.

To create a more realistic approach to bug classification, we determined that certain defects should be considered irrelevant to the project on which analysis is applied. That is, when an analysis is applied to a large project that is in production or time-sensitive, not every defect would necessarily be fixed. As such, we made the assumption that because JEdit is a standalone application that uses few resources and is not necessarily reliant on fast, consistent performance, resource leaks were irrelevant.

The results of our findings are summarized in the Figure 1 below. A significant majority (74%) of defects were true positives. Of the true positive results, a significant majority (78%) of defects were determined to be relevant.



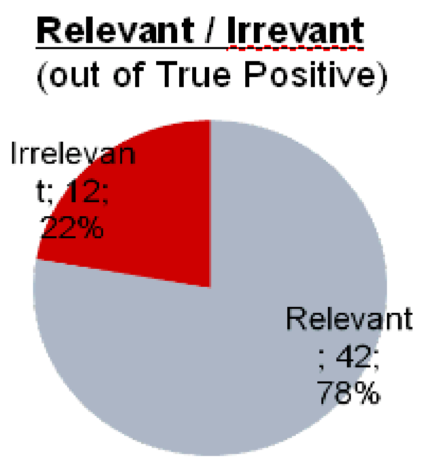
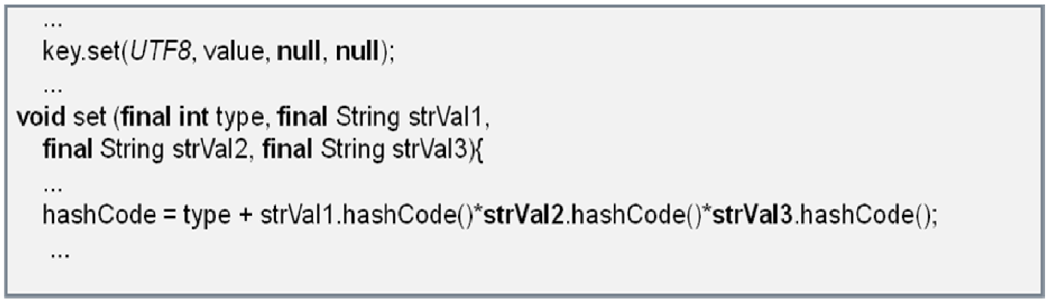


Figure 1: Prevent's reported false positive and true positive results

On 80K lines of code, analysis took 17 minutes. Considering the depth of analysis and the accuracy of the results, this is certainly acceptable performance.

The majority of true, relevant defects identified by Prevent were related to potential null dereference.

> *For example, In the ClassWriter class, a call is made to the set method of an Item object. Literal null values are passed as the third and fourth arguments.In the definition of set, the third and fourth arguments (strVal2 and strVal3, respectively) are of type String. No checks are made for potential null values passed to the function. Both strVal2 and strVal3 are dereferenced as null.*

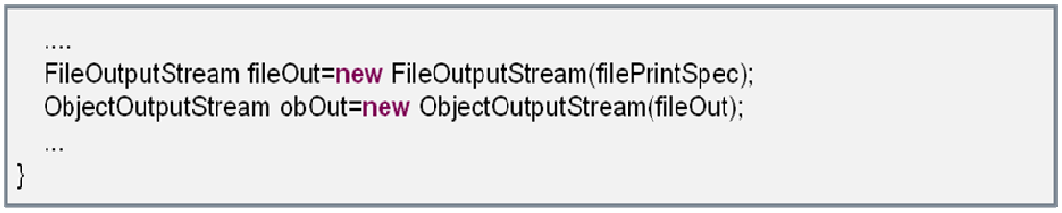


```
   ...
   key.set(UTF8, value, null, null);
   ...
void set (final int type, final String strVal1,
   final String strVal2, final String strVal3){
   ...
   hashCode = type + strVal1.hashCode()*strVal2.hashCode()*strVal3.hashCode();
    ...
```

[FORWARD_NULL] strVal2, strVal3 dereferenced

It was assumed that a company with time or resource limitations would not necessarily want to correct every true defect identified by Prevent. **We decided to classify resource leaks as irrelevant.**

The fileOut and obOut streams are created and used but are never released.

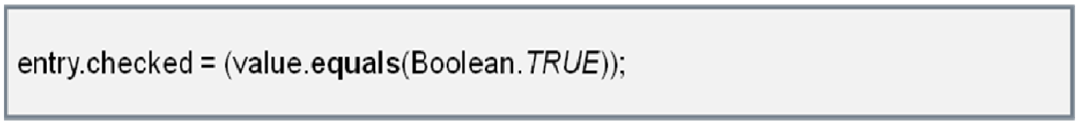


```
   ....
   FileOutputStream fileOut=new FileOutputStream(filePrintSpec);
   ObjectOutputStream obOut=new ObjectOutputStream(fileOut);
   ...
}
```

[RESOURCE_LEAK] return_without_free

Finally, the identified false positives were mostly found by the BAD_EQ checker. For these false positives, we determined that in a given context, the correct comparison was used.

The value is being compared for equivalence, not identity.



```
entry.checked = (value.equals(Boolean.TRUE));
```

[BAD_EQ] use_ref_eq

Lessons Learned

Prevent is a moderately easy application to use. The interface in the Eclipse plugin is intuitive and consistent with many other applications. However, the messages are sometimes cryptic. An understanding of the purpose and assumptions made by each checker is necessary in order to correctly triage defects.

Given the accuracy and precision of the results, the performance was acceptable. The analysis was consistently stable. 17 minutes for 80K lines of code is reasonable. For larger code bases, it is expected that quality assurance practices would be proportional. So, therefore, would be the acceptable length of analysis time. Furthermore, analyses are incremental. Thus, subsequent analyses will perform significantly faster.

Prevent is adequately modifiable. Individual checkers can be enabled and disabled. Plus they can be refined using the command prompt. The Eclipse plugin might be improved by including this fine-tuning in a dialog box.

While the tool is expensive, the cost can be justified by the application on which it is being applied. Critical applications certainly are well-suited for the depth of analysis that Prevent offers. It is unlikely that a small startup would benefit from this large of an investment, especially with the availability of free tools. Beyond the hard cost, there is a learning curve in applying Prevent. Knowledge of the foundational theory of the checkers is helpful. Understanding of their implications is essential.

Prevent requires a solid background in the language on which analysis is being applied. It is not intended as a learning tool and can most effectively be used by experienced developers. Evaluation of false and true positives can be difficult due to the interprocedural nature of many of the identified defects.

3. FindBugs

Overview

FindBugs is a static analysis tool to find defects in Java projects. It is free software and distributed under the terms of the Lesser GNU Public License. The name FindBugs is trademarked by The University of Maryland. Also, FindBugs is sponsored by Fortify Software

FindBugs is a popular analysis tool, and many companies and organizations are using the tool. As of December, 2007, FindBugs has been downloaded more than half a million times, according to the website [2].

FindBugs is free software, available under the terms of the Lesser GNU Public License. FindBugs was originally developed by Bill Pugh and David Hovemeyer[1]. It is maintained as an open source project. The trademark was registered by The University of Maryland, and the software is sponsored by Fortify Software.

FindBugs uses the concept called bug patterns. A bug pattern is a code idiom that often results in an error. The static analysis inspects Java bytecode to find occurrences of bug patterns. It is a static analysis tool i.e. FindBugs can find defects by simply inspecting a program's code, and that executing the program is not necessary. In addition, FindBugs works by analyzing Java bytecode, so it does not required even the program's source code to use it. However, as a downside, the nature of analysis tends to be sometimes imprecise, FindBugs can report many false warnings.

Following are some of the categories of defects reported by FindBugs[2].

- Correctness bug

   - “Probable bug - an apparent coding mistake resulting in code that was probably not what the developer intended. We strive for a low false positive rate. “
- Bad Practice
   - “Violations of recommended and essential coding practice. Examples include hash code and equals problems, cloneable idiom, dropped exceptions, serializable problems, and misuse of finalize. We strive to make this analysis accurate, although some groups may not care about some of the bad practices. “
- Dodgy
   - “Code that is confusing, anomalous, or written in a way that leads itself to errors. Examples include dead local stores, switch fall through, unconfirmed casts, and redundant null check of value known to be null. More false positives accepted. In previous versions of FindBugs, this category was known as Style. “

Experiment Setup

The FindBugs Eclipse plug-in has a property dialogue, in which the user can enable or disable specific detector, change the report configuration, and apply the XML-written filter file. In order to support a reasonable mapping between defects discovered by both Coverity Prevent and Findbugs, all of the detectors were enabled during analysis on JEdit. However we used the high priority settings which reported only high priority defects in order to reduce the time it would take us to analyze the false positives and true positives. With medium priority settings FindBugs reported around 340 defects and even more with low priority setting.

Experiment Results

In total 63 defects were identified by FindBugs on JEdit, as shown in Figure 2. A significant majority (90%) of defects were true positives. Of the true positive results, a significant majority (75%) of defects were determined to be relevant.

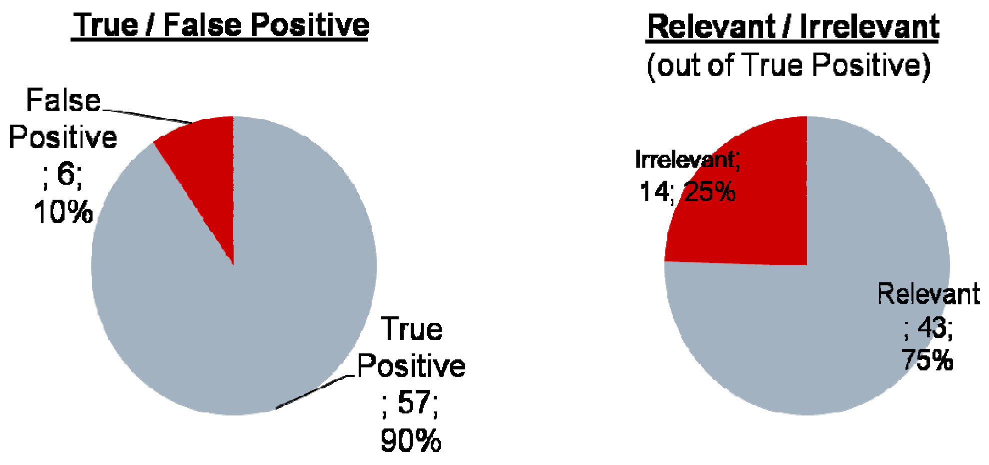


Figure 2: FindBug’s reported false positive and true positive results

The nature of these defects varied over the categories. 32 defects were found by the MALICIOUS_CODE detector, which flags malicious bug patterns such as a constant field which does not declared as final. 11 defects were found by the BAD_PRACTICE detector, which flags bad coding practices such as one usage of a random function. 11 defects were found by the STYLE detector, which flags bad coding styles. 6 defects were found by the CORRECTNESS detector, which flags the correctness defects such as possible NULL on the path. 2 defects were found by the PEFORMANCE detector, which flags performance issues. 1 defect was found by the MT_CORRECTNESS detector, which flags a problematic multi thread usage.

Evaluation of these results took approximately 2 person-hours to determine the validity of all 63 defects. Averagely, 2 minutes per a bug. However, as the analyzer goes on the analysis, the spent time per a bug was decreasing mainly because the user had been educated by the tool and the category given by the tool. To create a more realistic approach to bug classification, as we mentioned in Prevent section, we determined that certain defects should be considered irrelevant to the project on which analysis is applied. The details will be talked in the comparison section.

One example of true, relevant defects identified by FindBugs was about the abs function.

```
tipToShow = Math.abs(new Random().nextInt()) % count;
```

[RV] Bad attempt to compute absolute value of signed 32-bit random integer

This code generates a random signed integer and then computes the absolute value of that random integer. If the number returned by the random number generator is Integer.MIN_VALUE, then the result will be negative as well (since Math.abs(Integer.MIN_VALUE) == Integer.MIN_VALUE).

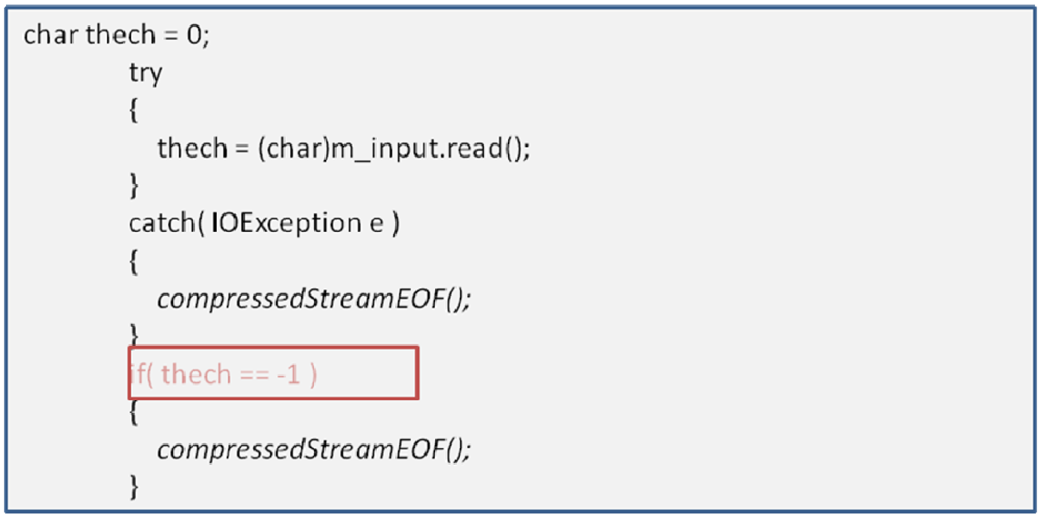
Another example for true, irrelevant defects was about a dead store to a local valuable. We decided to classify this as irrelevant because it might be possible to use the local valuable later and also it does not affect the JEdit quality attributes.

```
if(wrapMargin != 0.0f)
        this.wrapMargin = wrapMargin += 2.0f;
else
        this.wrapMargin = 0.0f;
```

[DLS] Dead store to local variable

This instruction assigns a value to a local variable, but the value is not read or used in any subsequent instruction. Often, this indicates an error, because the value computed is never used. Note that Sun's javac compiler often generates dead stores for final local variables. Because FindBugs is a bytecode-based tool, there is no easy way to eliminate these false positives.

The example of identified false positives was about the bad comparison of nonnegative value with negative constant. FindBugs can only do an intra-procedural analysis and cannot detect the return value of the function. In this example, FindBugs cannot detect the possibility that the local variable can have a negative value from a return value of a function. We categorized this case as a false positive.



```
char thech = 0;
          try
          {
             thech = (char)m_input.read();
          }
          catch( IOException e )
          {
             compressedStreamEOF();
          }
          if( thech == -1 )
          {
             compressedStreamEOF();
          }
```

[INT] Bad comparison of nonnegative value with negative constant

This code compares a value that is guaranteed to be non-negative with a negative constant.

Lessons learned

FindBugs is an easy and intuitive tool to use. The Eclipse plug-in interface is consistent with the Eclipse user interface, and the user familiar with Eclipse will not have any difficulty to use the tool. It is a good educational tool for Java beginners because the result of analysis tells many rules of thumbs about Java coding. Installing the Eclipse plug-in was also very easy. We just need to put the download URL into the Eclipse setting.

The approach of analysis is the concept of bag patterns, and the tool realizes the approach as a practical tool. The analysis time was considerably short. The result might include many false positives, but in practical the rate of false positives are less than 50%, according to [1]. We used the high-priority setting for our analysis on JEdit, and the rate of false positive was 10%.

The modifiability of Eclipse plug-in is also good. Individual detectors can be enabled and disabled. The filter can be defined by XML and it can be shared across the organizations. One downside of Eclipse plug-in is that it does not support exporting the result to Excel or CVS formats. However, we can copy the internal XML formatted result in the Eclipse directory, and import it from the Excel XML import wizard.

The license fee of the tool is zero. This tool will be helpful for small and/or start-up companies without fund for tool. To utilize this tool, developers might need some learning curves and also the analysis of the result might need additional time to the development lifecycle. However, the bug patters are good to know for Java developers and the analysis time will be good educational opportunities. Overall, we definitely recommend this tool to Java developers and we will use FindBugs in our Studio project.

4. Tool Comparison

This section will cover the qualitative as well as quantitative comparison of the two tools – Coverity Prevent and FindBugs.

Mapping between the categories of defects

Both tools provide a set of categories of the defects. We tried to create a mapping between such *categories* between the two tools as shown in the Table 1.

FindBugs	Coverity Prevent
• Bad practice	• *BAD_EQ* • *CALL_SUPER* • *PRIVATE_COLLECTION* • *UNMODIFIABLE_COLLECTION*
• *Correctness*	• *CLASSCAST* • *FORWARD_NULL* • *NULL_RETURNS* • *REVERSE_NULL* • *MUTABLE_HASHCODE* • *MUTABLE_COMPARISON*
• • *Multithreaded correctness*	• *LOCK_INVERSION* • *LOCK_ORDERING* • *DOUBLE_CHECK_LOCK* • *INDIRECT_GUARDED_BY_VIOLATIO N* • *GUARDED_BY_VIOLATION* • *UNSAFE_LAZY_INIT*
• *Performance*	• *RESOURCE_LEAK*
• *Security* • *Malicious code vulnerability*	• *JDBC_CONNECTION*
• *Dodgy*	
• *Internationalization*	

Table 1: Mapping categories of defects found by FindBugs and Prevent

Quantitative analysis

Mapping created in the last section helps analyze the differences between kinds of defects that these tools report. We ran the tool over the source code of JEdit which has more than 80K lines of code. The statistics are presented in the Table 2.

	FindBugs	Coverity Prevent
Settings I (Enabled priority)	Only high priority defects	Default Setting
Settings II (Enabled category of defects)	• Bad Practice • Performance • Correctness	**Note**: *Corresponding categories of defects from the above mapping table.*

	• Multithreaded correctness	
Application on which the tool is run	JEdit (88 KLOC)	JEdit (88 KLOC)
Total defects found	52	89

Table 2: Defects reported by FindBugs and Prevent with customization

One interesting thing to note is, ***none*** *of the defects reported are originated from the same line.* Table 3 and Figure 3 provide are some details about the defects reported in each category:

	FindBugs	**Coverity Prevent**
Bad Practice	10	23
Performance	2	11
Correctness	39	45
Multithreaded Correctness	1	10
	52	89

Table 3: Defects reported by FindBugs and Prevent under different category

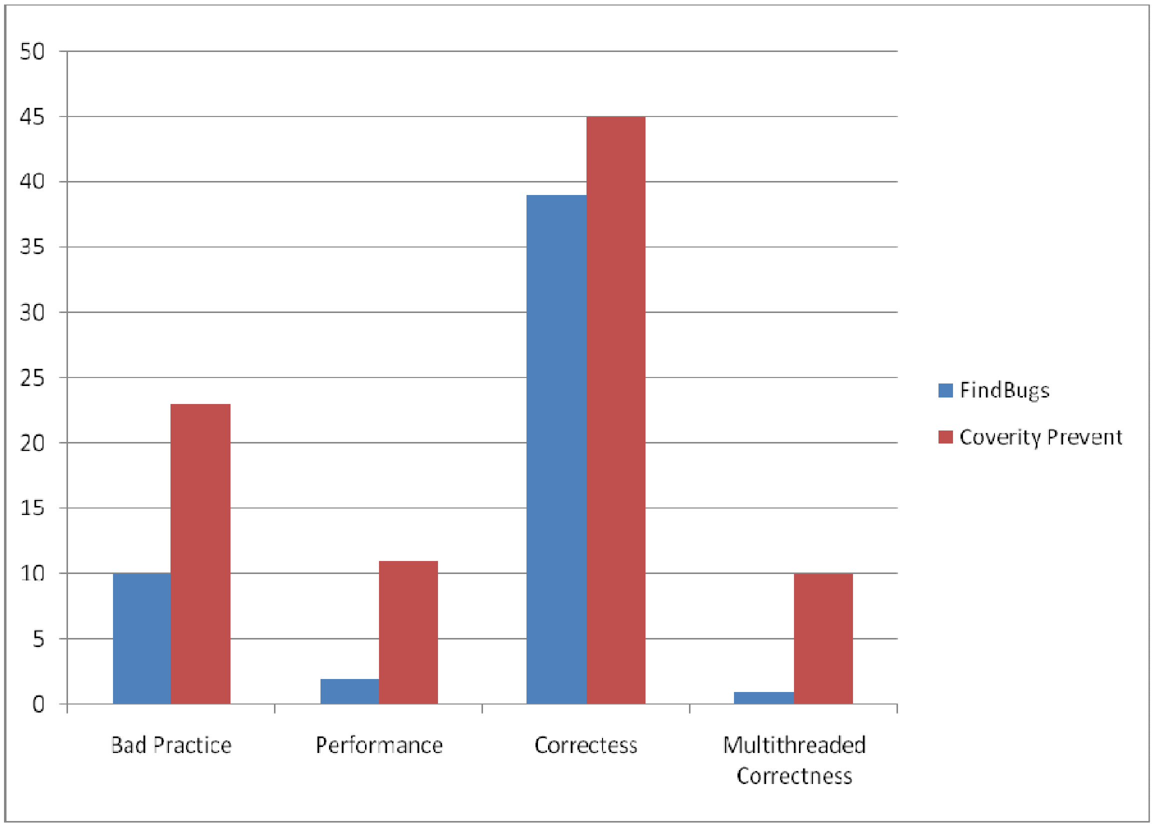


Figure 3: Defects reported by FindBugs and Prevent under different category

If we were to run the FindBugs with all the priorities set, the tool will output 401 defects and 60 of them are *false positives* which are very high as compared to only 19 that Coverity Prevent has reported.

Analysis Time

We ran both the tools over JEdit source code. The source has more than 80K lines of code. Table 4 contains the results.

	Coverity Prevent	**FindBugs**
JEdit source code	17 min	6 min

Table 4: Analysis time taken by Coverity Prevent and FindBugs

As the results shows, FindBugs ran over the code much faster than Coverity Prevent. But Coverity Prevent is performing more complex operations (Interprocedural analysis, applying boolean satisfiability solvers, etc.) and finding more critical defects. Coverity Prevent takes extra time to reduce the *false positives* as well.

Qualitative Differences

Here are few qualitative differences between the two tools:

Analysis techniques

FindBugs:

- One of the main techniques FindBugs uses is to syntactically match source code to **known suspicious programming practice.**
  - FindBugs checks that calls to wait(), used in multi-threaded Java programs, are always within a loop—which is the correct usage in most cases.
- It uses **dataflow analysis** to check for defects.
  - For example, FindBugs uses a simple, intraprocedural (within one method) dataflow analysis to check for null pointer dereferences.

Coverity Prevent:

- Coverity Prevent syntactically matches the source code to **known suspicious programming practice.**
- It uses Interprocedural analysis
- It uses data flow analysis
- It applies Boolean satisfiability solvers
- Coverity Prevent claims to use *False Path Engine* (Solves each branch condition to determine if it will be true, false, or unknown on the current path) to efficiently remove obvious false positives from the set of defects reported.

Configurability

- FindBugs is easy to install and configure, and provides numerous options including use of *regular expression* to customize the kind of defects reported.
- Coverity Prevent is easy to install and configure for Java, but configuring it for C/C++ is a bit tricky and time consuming. It supports customization where the user can select the kind of errors he/she wants the tool to report.
- Relatively, Configurability provided by FindBugs is better than Coverity Prevent

Tool price

- Coverity Prevent is a commercial product ($50000 for 500,000 lines of code) while FindBugs is a free ware.

Usability

- Both the tools are intuitive to use, and should not cause any usability problems for the general **eclipse** platform users.
- The report view for both the tools lists the defects serially, but do not categorize them. This is something desirable as a user.
- Coverity Prevent provides its own Defect manger engine. Engine provides a web based application to analyze and administer defects.

Observation

Following are few key observations about the tools

FindBugs

- Because its analysis is sometimes imprecise, FindBugs can report *false warnings*, which are warnings that do not indicate real errors.
- FindBugs generated far too many errors/warnings and review all of them manually is difficult.

Coverity Prevent

- It has total path coverage.
- It has low false positive rate.
- It finds defects which are difficult to locate by mere review.

5. Conclusion

Some types of defects reported by the tool are of interest to us while some are not. These tools have some default settings; however these settings can be changed to suit the needs of our analysis. It is important to identify the type of defects we are interested in and customize the tools accordingly.

Analyzing false positives is time consuming and unproductive. Different static analysis tools report different rates of false positives and true positives which can vary significantly. The tools that report fewer false positives can be considered more accurate than tools that don’t, however, they may take longer analysis time. Thus, there is a tradeoff between accuracy and analysis time.

Inter-procedural analysis techniques can capture errors that even formal reviews can miss easily. Static analysis is also helpful in correcting bad coding practice besides detecting mechanical defects such as security or performance defects.

Static analysis tools report several kinds of defects. For example, Prevent could not detect internationalization defects. Since different tools find different kinds of defects, it is our recommendation that more than one tool be used to analyze the projects.

6. References

[1] Infoworld article, http://www.infoworld.com/article/06/01/26/73919_05FEcode_1.html
[2] FindBugs website, http://findbugs.sourceforge.net/users.html