Which SStuBs are missed by Static Analyzers?

Nazmus Sakeef
sakeef@ualberta.ca
University of Alberta
Edmonton, Alberta, Canada

Sakib Hasan
sakib2@ualberta.ca
University of Alberta
Edmonton, Alberta, Canada

ABSTRACT

One of the most essential tasks of software quality assurance is bug identification and prevention. Many of the key issues that developers face today can be discovered and even fixed whole or partially using automated bug detection tools. Recent research has discovered that there are a significant number of basic but very aggravating errors in code bases that are easy to rectify but difficult to notice because they do not significantly impair the operation. Such errors are often introduced by programmers unintentionally, as a result of inattention. We were looking for information on things like: How effective are static analyzer tools in detecting these simple bugs? Furthermore, what are the most typical bugs that analyzers overlook, and why? Is there any way to make the analyzer tools better? This work addresses these questions by applying three widely used static bug analyzer tools on the *ManySStuBs4J* dataset, which comprises many basic, stupid bugs reported in GitHub projects implemented in the Java programming language. The findings show that: (i) the tools can only detect a limited number of categories of simple stupid bugs, with good performance in detecting those categories; (ii) aside from the selected categories, other categories are largely missed due to domain-specific issues and formats that the tools do not support; and (iii) The tools' performance in finding simple stupid bugs can be increased by a certain percentage. These findings aid potential users in evaluating the efficacy of such tools, as well as encouraging further improvement of the tools in future in simple stupid bug detection.

KEYWORDS

Simple Stupid Bugs, SStuB, Static Analyzer Tools, Bug Identification

1 INTRODUCTION

Fast development is a must in today's world of rapidly changing expectations and needs for software. However, there is a price to pay: bugs are unavoidable. For most developers, finding and correcting them has become an almost daily task. As a result of the importance and difficulty of this duty, many bugs go unreported for long periods of time, with many flaws only being discovered after they have had a negative impact. In order to avoid this in a key piece of software, extensive testing is required to assure proper functionality. Since, the source code of such software systems is produced by humans, released software may still have multiple bugs despite extensive testing and code reviews.

We know about various bugs, however we may not have heard about stupid bugs. Locating and extracting these bugs both before and after repair might be time consuming, yet their remedies are so simple that many developers dismiss them as "stupid" after realization [1]. One of the most important responsibilities in ensuring software security is identifying vulnerabilities in the source code. To examine these defects and logical inconsistencies in codes, static analysis methodologies and static analyzer tools are available. Though these software analyzer tools are capable of detecting significant code flaws, they fall behind in case of simple stupid bugs. There are some prior research works [1], [2] with Static Analyzer tools in detecting SStuBs. The works concluded that static analyzers faced difficulty finding these bugs, however an elaborate explanation on the possible reasons was missing. Furthermore, no effort has been discovered that improves the effectiveness of analyzer tools in detecting SStubs.

To gain a deeper understanding about these issues, we analyzed SstuBs in the ManySStuB4J dataset [1], which is one of the largest public datasets comprising SStuBs from open-source Java projects. There are around 153, 652 single statement bug-fix data from organised into 16 templates. Furthermore, we studied the efficiency of three static analyzer tools, SonarQube, Spotbugs, and PMD, in finding SStuBs. In addition, we tried to figure out which bugs the analyzers are most likely to miss. Finally, we investigated the analyzer tools to see if there's anything we can do to improve the analyzers' performance in finding SStuBs. Our investigation was broken down into the following research questions:

- *RQ1:* How effective are the static analyzers in detecting SStuBs?
- *RQ2:* Which categories of SStuBs are commonly undetected among the analyzers?
- *RQ3:* Does improving an analyzer affect the overall performance in detecting *SStuBs*?

We used three static analyzers, SonarQube [*], SpotBugs [†], and PMD [‡] on 100 maven and 100 top-ranked Java projects from the repository. We collected the theoretical promises of the analyzer tools by going through their documentation and source code. Then, we collected bug information and compared the results upon scanning the projects with the analyzers' theoretical findings. From the findings, we sorted out the most missed bugs by the analyzer tools. Next, we selected two categories of bugs to improve upon manually as none of the analyzer tools provide no guidelines for improving the tools from outside. Finally, we concluded that performance improvement for the tools in detecting SStuBs is possible if we can integrate improvement in the code base of any static analyzer. In summary, the contributions of this paper are as follows:

- We analyzed performance of different Static Analyzers on *ManySStuBs4J* dataset for capturing simple stupid bugs (SStuB).
- We employed SonarQube, which has not been used before to detect *SStuB*.
- We manually implemented a scanner for detecting two *SStuBs* new SStuB patterns.

[*]https://www.sonarqube.org/
[†]https://spotbugs.github.io/
[‡]https://pmd.github.io/latest/pmd_rules_java.html

- We also showed that improvement of the static analyzers is possible in detecting *SStuBs*.

2 BACKGROUND KNOWLEDGE

In software maintenance, bug discovery and program repair are essential. Detecting and correcting issues early in the software development cycle lowers the cost of software maintenance. Bugs can be found in single statements or can be found in a block of code. And *Simple Stupid Bugs* [1] are those whose fixes are just a little tweak.

Simple Stupid Bugs: Simple Stupid Bugs are rarer than other types of bugs, making them more difficult to locate. SStuBs or simple stupid bugs are minor inaccuracies that do not prevent the code from running or compiling. SStuBs appear on a single statement, and the remedy for that statement is also contained within that statement. However, influencing the behavior of codes in a subtle way might have unforeseen repercussions. Some examples of SStubs are given in *Figure 1.*

.

Static Analysis Method: Static code analysis is a method of debugging that involves reviewing source code prior to running a program. It is accomplished by comparing a set of code against a set (or several sets) of coding rules. Along with source code analysis, the terms static code analysis and static analysis are frequently interchanged. This type of investigation looks for flaws in source code that could lead to vulnerabilities. This can, of course, be accomplished by manual code reviews. However, utilizing Static Analyzers is far more efficient as these can be easily integrated into IDE. Such tools can aid in the detection of flaws in software development.

Given a set of bugs "Bugs" and static code analyzer tools. A detected bug is represented as a tuple (bg, warn), with bug being the first element where $bg \in Bugs$ is a bug, while "warn" is a warning that indicates to a bug in the code. A single bug, "bg" may be identified by numerous warnings, such as (bg, warn_1) and (bg, warn_2), and a single warning, such as (bg_1, warn) and (bg_2, warn), may indicate to multiple bugs [3]. Applying the tool to the buggy version of the code and manually inspecting each issued warning is a simplistic technique to determining whether a tool finds a specific bug.

When opposed to uncovering vulnerabilities later in the development cycle, Static analyzer tools' feedback can save time and effort. These tools can be used with a wide range of software and can be used repeatedly [4]. There are many tools and strategies for automatically detecting bugs have been developed. Among them SonarQube, SpotBugs, and PMD are common in order to reduce the manual labour of discovering and repairing bugs [5].

SonarQube: SonarQube enables all developers to build cleaner and safer code. This tools enhances the development workflow and advise teams with thousands of automatic Static Code Analysis rules in more than 25 programming languages and direct integration with the DevOps platform. SonarQube integrates with existing tools and give alerts when codebase's quality or security is jeopardized.

SpotBugs: SpotBugs is a software that looks for problems in Java code using static analysis. More than 400 insect patterns are checked by SpotBugs. SpotBugs can be used independently or with a variety of integrations, such as Ant, Maven, Gradle, and Eclipse.

PMD: PMD is a open-source code analyzer that supports C/C++, Java, and JavaScript. This is a straightforward tool that can be used to identify common flaws like unused variables, empty catch blocks, unnecessary object creation, and so forth. In Java, it also finds duplicate code.

However, traditional static analysis methods, require professionals to develop particular detection algorithms for different sorts of defects in advance, especially for SStuBs, which is still time-consuming and labor-intensive, and may result in high false positive or negative rates.

3 RELATED WORKS

Although there is a large body of work on software bugs and their discovery by static analyzer tools, finding SStuBs from the ManySStuBs dataset using static analyzer tools is a new concept.

Karampatsis et. al. [1] investigated whether a static analyzer (SpotBugs) could detect SStuBs and discovered only 12% of them were flagged. Additionally, they discovered 200M potential issues, rendering static analyzers useless for detecting SStuBs. However, they did not provide any explanations for this failure. One of the significant limitations, according to the study, is that a developer would have to sift through hundreds of thousands of SpotBugs warnings to find a single SStuB.

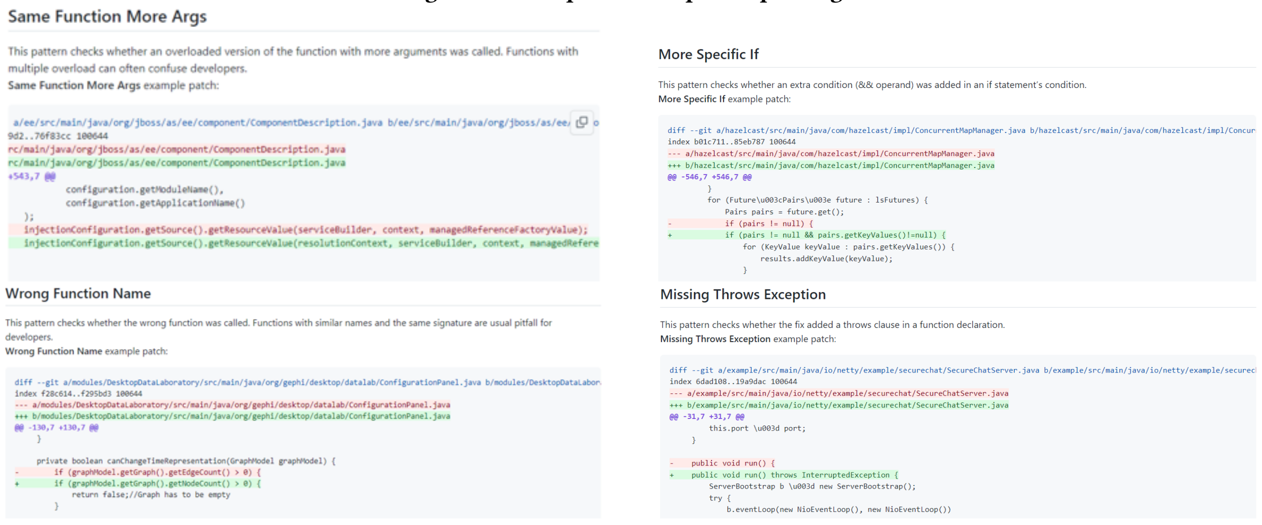
Mosolygo, B. et. al. [2] tried another static analyzer, PMD, to check if it could identify any lines containing SStuBs, but it failed to do so for the studied dataset. The authors concluded that static analyzers had difficulty finding these bugs. Although they did not elaborate on why that happened.

Another research by Hua, J. et. al. [6], examined how well deep learning-based bug detection approaches perform when it comes to finding SStuBs. They re-implemented two cutting-edge approaches in about 3,000 lines of code and applied them to the detection of Java SStuBs. Experiments on large-scale datasets show that, while deep vulnerability detectors can outperform conventional static analyzers, SStuBs are difficult to discover when compared to traditional complex vulnerabilities. The paper's findings suggest that diverse techniques should be coupled for better detection of SStuBs, and the research has practical relevance in this regard.

Another study by Habib, A. et. al. [3] examined how many real-world bugs are detected with static bug detectors. The authors looked at the effects of using three popular static bug detectors on an enlarged version of the Defects4J dataset [7], which contains 15 Java projects with 594 documented flaws. The authors employed a novel methodology that combines an automatic analysis of warnings and bugs with manual evaluation of each candidate of a detected bug to determine which of these bugs the tools find. The study's findings revealed that static bug detectors can discover a significant number of all bugs, and that different techniques are typically complementary to one another. The researchers also discovered that the vast majority of the flaws analysed are missed by conventional code analyzers.

Tomassi, D. A. [8], presented a method for determining the effectiveness of static analyzer tools in detecting defects in real-world software. The authors provided a preliminary analysis on *Error-Prone* and *SpotBugs*, two prominent static analyzers. They looked at 320 real Java bugs from the *BugSwarm* dataset [**?**] to explore

Figure 1: Examples of Simple Stupid Bugs



Same Function More Args

This pattern checks whether an overloaded version of the function with more arguments was called. Functions with multiple overload can often confuse developers.

Same Function More Args example patch:

```
 a/ee/src/main/java/org/jboss/as/ee/component/ComponentDescription.java b/ee/src/main/java/org/jboss/as/ee/ o
9d2..76f83cc 100644
rc/main/java/org/jboss/as/ee/component/ComponentDescription.java
rc/main/java/org/jboss/as/ee/component/ComponentDescription.java
+543,7 @@
            configuration.getModuleName(),
            configuration.getApplicationName()
   );
   injectionConfiguration.getSource().getResourceValue(serviceBuilder, context, managedReferenceFactoryValue);
   injectionConfiguration.getSource().getResourceValue(resolutionContext, serviceBuilder, context, managedRefere
```

Wrong Function Name

This pattern checks whether the wrong function was called. Functions with similar names and the same signature are usual pitfall for developers.

Wrong Function Name example patch:

```
diff --git a/modules/DesktopDataLaboratory/src/main/java/org/gephi/desktop/datalab/ConfigurationPanel.java b/modules/DesktopDataLabor
index f28c614..f295bd3 100644
--- a/modules/DesktopDataLaboratory/src/main/java/org/gephi/desktop/datalab/ConfigurationPanel.java
+++ b/modules/DesktopDataLaboratory/src/main/java/org/gephi/desktop/datalab/ConfigurationPanel.java
@@ -130,7 +130,7 @@
     }

     private boolean canChangeTimeRepresentation(GraphModel graphModel) {
-        if (graphModel.getGraph().getEdgeCount() > 0) {
+        if (graphModel.getGraph().getNodeCount() > 0) {
             return false;//Graph has to be empty
         }
```

More Specific If

This pattern checks whether an extra condition (&& operand) was added in an if statement's condition.

More Specific If example patch:

```
diff --git a/hazelcast/src/main/java/com/hazelcast/impl/ConcurrentMapManager.java b/hazelcast/src/main/java/com/hazelcast/impl/Concur
index b01c711..85eb787 100644
--- a/hazelcast/src/main/java/com/hazelcast/impl/ConcurrentMapManager.java
+++ b/hazelcast/src/main/java/com/hazelcast/impl/ConcurrentMapManager.java
@@ -546,7 +546,7 @@
         }
         for (Future\u003cPairs\u003e future : lsFutures) {
             Pairs pairs = future.get();
-            if (pairs != null) {
+            if (pairs != null && pairs.getKeyValues()!=null) {
                 for (KeyValue keyValue : pairs.getKeyValues()) {
                     results.addKeyValue(keyValue);
                 }
```

Missing Throws Exception

This pattern checks whether the fix added a throws clause in a function declaration.

Missing Throws Exception example patch:

```
diff --git a/example/src/main/java/io/netty/example/securechat/SecureChatServer.java b/example/src/main/java/io/netty/example/securec
index 6dad108..19a9dac 100644
--- a/example/src/main/java/io/netty/example/securechat/SecureChatServer.java
+++ b/example/src/main/java/io/netty/example/securechat/SecureChatServer.java
@@ -31,7 +31,7 @@
         this.port \u003d port;
     }

-    public void run() {
+    public void run() throws InterruptedException {
         ServerBootstrap b \u003d new ServerBootstrap();
         try {
             b.eventLoop(new NioEventLoop(), new NioEventLoop())
```

which ones may be detected by the analyzers and how many were actually detected. They discovered that ErrorProne and SpotBugs can detect 30.3% and 40.3% of the bugs respectively. The analyzers are effortless to integrate into the tool chain of various projects that use the Maven build system. The tools, on the contrary, are not as satisfactory at discovering the problems, with only one bug being correctly spotted by SpotBugs.

Mashhadi, E. et. al. [9], presented CodeBERT, a transformer-based neural architecture pre-trained on a huge corpus of source code, as the basis for a unique automated programme repair approach. The authors fine-tuned the model using the ManySStuBs4J small and large datasets. Their technique correctly predicted the fixed codes implemented by developers in 19-72% of cases, depending on the kind of datasets. They also claimed that their method can cure a variety of bug kinds, even if there are only a few examples of those bug categories in the training dataset. And all these previously carried out experiments and their findings motivated us to further investigate on the points that we addressed in this study.

4 DATASET

The ManySStuBs4J dataset [1] is a set of simple Java bug fixes that can be used to test program repair techniques. The dataset contains simple statement bugs extracted from open-source Java projects on GitHub, which are categorized into one of 16 syntactic patterns known as SStuBs if possible. The dataset comes in two versions. One was mined from the top 100 Java Maven projects, while the other was mined from the top 1000 Java projects. Table 1 shows the number of simple bugs in maven and java projects respectively. The dataset's authors maintained only bug commits with single statement changes, disregarding stylistic variations. Refactorings such as variable, function, and class renaming, function argument renaming, and modifying the number of arguments in a function were not included in the dataset. In a separate JSON file, any problems that suit one of 16 patterns are also labeled with whose pattern(s) they fit.

Table 1: Dataset Statistics [1]

Projects	SStuBs
100 java maven	7824
1000 java	51537

While dealing with dataset, we faced some difficulties in running and testing all the 1000 java programs. Memory and runtime issues for 1000 java programs as well as batch processing were some of them while running on static analyzers. For this reason, we selected 100 maven projects and 100 java projects from a pool of 1000. We chose 100 java projects using the ranking of the projects considering number of forks and watchers of the project. The selected 100 maven and java project will be found in this GitHub *repo*.

5 METHODOLOGY

This section outlines our approach to identifying and collecting selected categories of bugs after running the static analyzer tools. In addition, the section presents how the manual analyzer was implemented in detecting SStuBs. To begin, in Section 5.1, we discussed how the theoretical promises of the analyzer tools were collected and matched them with the defined 16 templates of *SStuB*s. In addition, how we identified and collected the matching bug information of the selected types are thoroughly described in Section 5.2. Finally, the implementation of a manual analyzer in detecting some more patterns of SStuBs is discussed in Section 5.3.

5.1 Collecting Theoretical Promises

To gain a deep understanding, we searched through the source code and documentation of each of the analyzers to find out theoretical promises for detecting bugs and other vulnerabilities. Then, we matched the rules against the *SStuB*s patterns mentioned in the dataset to measure the theoretical promises of the analyzers in capturing SStuBs. Figure 2 demonstrates snapshots of going through

Table 2: Theoretically promised bugs of analyzer tools and no of particular bugs in database

SStuBs Template	SonarQube	PMD	SpotBugs
More Specific If	Merge If..	Deeply Nested If..	-
Missing Throws Exception	Define and Throw a Dedicated Exception	Define and Throw a Dedicated Exception	DCN_NullPointer_Exception, REC_CatcException
Delete throws Exception	-	-	DCN_NullPointer_Exception, REC_CatcException
Promised	maven (291), Java (953)	maven (291), Java (953)	maven (116), Java (95)

the static analyzers documentation and collecting the bug details of relevant patterns.

Out of the 16 templates, only three of them matched with the theoretical promises of the analyzer tools. For PMD analyzer, "*Deeply Nested If*" and "*Define and throw a dedicated exception*" categories are matched against SStuBs patterns. For SonarQube analyzer, almost similar to the PMD, "*Merge IF*", and "*Define and throw exception rules*" were identified, and lastly, "*DCN_Null pointer exceptions*" and "*REC_Catch exceptions*" were matched in case of SpotBugs when we finished analyzing their documentation.

We developed a script to find how many of these three categories of bugs are present in our selected portion of the dataset projects. For PMD and SonarQube analyzer tools, we found there are 291 bugs in maven and 953 bugs in java projects of those promised categories were present. Both analyzers guaranteed to identify the same pattern of simple stupid bugs, therefore these metrics were identical. SpotBugs promised to identify a separate category of bugs, and the selected projects contain 116 bugs in maven and 95 bugs in java. Table 2 shows the theoretical promises of the analyzers in detecting particular SStuBs categories in selected projects. .

5.2 Collecting the promised bugs

After collecting the theoretical promises, we analyzed whether the tools kept the promises or not. We ran the analyzer tools on both 100 maven and 100 java projects. Then the bug reports generated by the analyzer tools with proper keywords were searched and collected the bugs. However, we did not select and collect all the bugs we found by the report. The reason behind this is there may be coincidental matching of a bug and a warning among the automatically determined candidates for detected bugs. Consider the case where a bug detector warns of a possible missing throw exception at a certain line, and this line is changed as part of a bug repair. If the repaired bug has nothing to do with the missed throw exception, the warning would not have helped a developer find it.

We manually verified all candidates for reported bugs and compared the warning messages to the buggy and corrected versions of the code to eliminate such coincidental matches. Each candidate was assigned to one of two groups: (i) This is a full match if the warning matches the fixed bug and the fix only changes lines that affect the flagged bug. (ii) If the fix has nothing to do with the warning message, it's a mismatch. The "mismatch" scenario was not considered in this study.

Unfortunately, static bug detectors generated a lot of warnings, and manually evaluating each one for each flawed version of a program did not scale to the amount of issues we're interested in. However, to locate the designated types of defects and warnings, we manually analyzed all of the issues and code smells.

Figure 3 and figure 4 show bug identification by SonarQube on java and maven projects respectively. The figures clearly demonstrate how we run SonarQube on the projects and collected bug information as well as how SonarQube detected the promised categories of Bugs "*Merge this if*" and "*Define and throw a dedicated exception*" correspondingly. We collected all the bugs information of the analyzer tools manually and reported them in *.csv* files separately. This manual bug detection and reporting took a long time, which is another reason for limiting the number of projects compared to the original dataset. .

5.3 Implementation of Manual Analyzer

Three of the analyzers theoretically and practically guaranteed to identify only three categories of bugs among the 16 bugs specified in the *ManuSStuBs4J* dataset, as shown in the table **??**. We tried to improve one of the analyzer tools in detecting SStuBs to check if the performance increased or not, in order to support our claim for our third research question.

Initially, we intended to explore the source code of the analyzer tools to see whether there was any room for improvement in the analyzer. SonarQube is a commercial product and that is not open-sourced, as we discovered. There are available codebases for SpotBugs and PMD, but no recommendations or information are offered for improving or contributing from the outside.

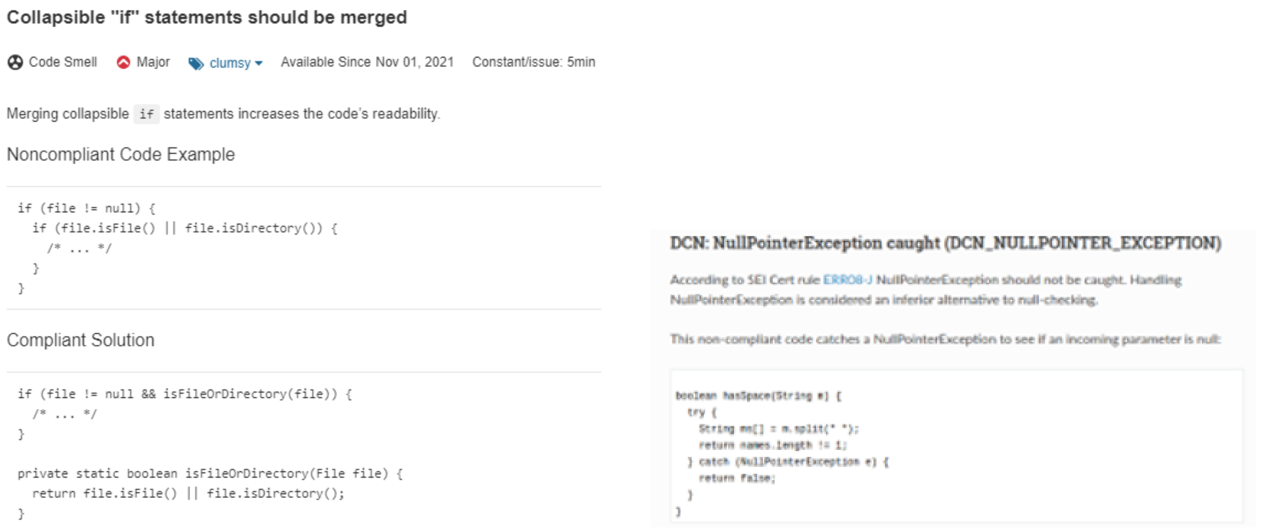
So, among the 16 defined bug templates from the dataset, we selected two templates "*Same Function More Arguments*" and "*Same Function Less Arguments*" to improve upon manually by implementing an analyzer. We chose these two categories of bugs since there were a significant number of bugs of those kinds present in the original dataset, 5100 and 1588 bugs respectively. This means that they were so frequent categories of faults developers made mistakes in their codes and hard to identify them.

As it was not possible to improve any of the analyzer packag as a whole, we chose to implement a scanner file in Python that can read a single java file and find out the selected types of bugs present in a Java file. The scanner output also shows the line number of the bug found as well as the expected number of arguments in the function. We scanned 16 Java files with our scanner by manually integrating bugs of the SStuBs patterns that we want to capture. The scanner and scanned Java files can also be found at this *repo*.

6 RESULTS

This section summarises the findings we got in order to address each of our research questions. The performance of the analyzer

Figure 2: Demonstration of Collecting Practical Promises



Collapsible "if" statements should be merged

Code Smell · Major · clumsy · Available Since Nov 01, 2021 · Constant/issue: 5min

Merging collapsible `if` statements increases the code's readability.

Noncompliant Code Example

```
if (file != null) {
  if (file.isFile() || file.isDirectory()) {
    /* ... */
  }
}
```

Compliant Solution

```
if (file != null && isFileOrDirectory(file)) {
  /* ... */
}

private static boolean isFileOrDirectory(File file) {
  return file.isFile() || file.isDirectory();
}
```

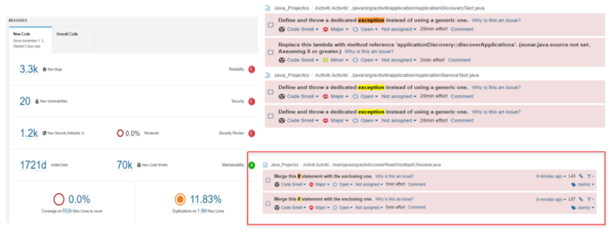
DCN: NullPointerException caught (DCN_NULLPOINTER_EXCEPTION)

According to SEI Cert rule ERR08-J NullPointerException should not be caught. Handling NullPointerException is considered an inferior alternative to null-checking.

This non-compliant code catches a NullPointerException to see if an incoming parameter is null:

```
boolean hasSpace(String m) {
  try {
    String ms[] = m.split(" ");
    return names.length != 1;
  } catch (NullPointerException e) {
    return false;
  }
}
```

Figure 3: SonarQube on Java Projects



Java_Projectss / Activiti.Activiti/.../java/org/activiti/application/ApplicationDiscoveryTest.java

Define and throw a dedicated exception instead of using a generic one. Why is this an issue?
Code Smell · Major · Open · Not assigned · 20min effort · Comment

Replace this lambda with method reference 'applicationDiscovery::discoverApplications'. (sonar.java.source not set. Assuming 8 or greater.) Why is this an issue?
Code Smell · Minor · Open · Not assigned · 2min effort · Comment

Java_Projectss / Activiti.Activiti/.../java/org/activiti/application/ApplicationServiceTest.java

Define and throw a dedicated exception instead of using a generic one. Why is this an issue?
Code Smell · Major · Open · Not assigned · 20min effort · Comment

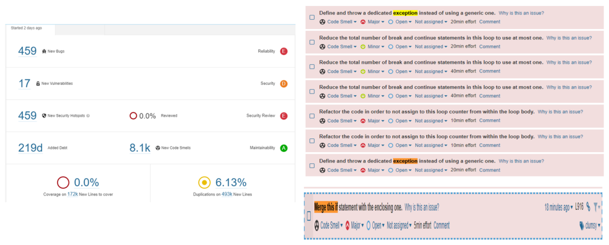
Define and throw a dedicated exception instead of using a generic one. Why is this an issue?
Code Smell · Major · Open · Not assigned · 20min effort · Comment

Java_Projectss / Activiti.Activiti/.../main/java/org/activiti/core/el/ReadOnlyMapELResolver.java

Merge this if statement with the enclosing one. Why is this an issue? 9 minutes ago · L43
Code Smell · Major · Open · Not assigned · 5min effort · Comment · clumsy

Merge this if statement with the enclosing one. Why is this an issue? 9 minutes ago · L57
Code Smell · Major · Open · Not assigned · 5min effort · Comment · clumsy

Figure 4: SonarQube on Maven Projects



Define and throw a dedicated exception instead of using a generic one. Why is this an issue?
Code Smell · Major · Open · Not assigned · 20min effort · Comment

Reduce the total number of break and continue statements in this loop to use at most one. Why is this an issue?
Code Smell · Minor · Open · Not assigned · 20min effort · Comment

Reduce the total number of break and continue statements in this loop to use at most one. Why is this an issue?
Code Smell · Minor · Open · Not assigned · 40min effort · Comment

Reduce the total number of break and continue statements in this loop to use at most one. Why is this an issue?
Code Smell · Minor · Open · Not assigned · 40min effort · Comment

Refactor the code in order to not assign to this loop counter from within the loop body. Why is this an issue?
Code Smell · Major · Open · Not assigned · 10min effort · Comment

Refactor the code in order to not assign to this loop counter from within the loop body. Why is this an issue?
Code Smell · Major · Open · Not assigned · 10min effort · Comment

Define and throw a dedicated exception instead of using a generic one. Why is this an issue?
Code Smell · Major · Open · Not assigned · 20min effort · Comment

Merge this if statement with the enclosing one. Why is this an issue? 18 minutes ago · L916
Code Smell · Major · Open · Not assigned · 5min effort · Comment · clumsy

tools in detecting SStuBs was shown in section 6.1 to answer our first research question. We next showed in section 6.2, which simple stupid bugs are usually overlooked by static bug analyzer tools. Finally, in section 6.3, the manual analyzer's performance in discovering two fixed patterns of SStuBs on several selected Java projects is shown.

6.1 Analysis of RQ1

Table 3, 4, 6, and 7, show the performance of the static analyzer tools in detecting three categories of SStuBs: *"More Specific If"*, *"Define and Throw Exception"*, and *"Null Pointer Exception"*.

Analysis by SonarQube Table 6 and 7 show the performance of SonarQube in detecting the promised categories of bugs. As SonarQube was not analyzed before in detecting SStuBs, we did some thorough analysis with SonarQube. We attempted to identify both the bugs of promised categories already present in the database and also the bugs SonarQube detected. Furthermore, we demonstrated how many of the bugs found by SonarQube corresponded to the bugs defined in the dataset.

In Table 6, we can see that, for the *"Activi.activi"* java project, there were 9 *"Merge_If"* bugs present in the dataset and 0 *"Exception"* bugs. SonarQube also detected 9 bugs of *"Merge_if"* type. However in case of *"Exception"* bugs, SonarQube detection 23 bugs, while there were none in the dataset. One thing to note is that, though SonarQube detected the same amount of *"Merge_If"* bugs, only two of them matched the dataset out of nine. Again, for the *"alibaba.canal"* project, there were 5 *"Merge_If"* bugs and 2 *"Exception"* bugs present in the dataset. SonarQube, on the other hand, detected 18 bugs of *"Merge_if"* type and 8 of *"Exception"* type. Though SonarQube detected much more bugs of the selected categories than defined by dataset, out of 18 *"Merge_If"* bugs, only 1 matched with the database and the number was 0 for *"Exception"* category.

SonarQube showed similar properties in finding SStuBs in maven projects, as seen in table 7. In the database for the *"netty.netty"* project, there were 12 *"Merge If"* bugs and 3 *"Exception"* bugs. However, SonarQube detected a much more bugs of *"Merge_if"* type and that was 46. On the other hand, for the *"Exception"* type bugs, SonarQube detected 9. Still, we saw the similar characteristics that there were mismatches between the results of SonarQube and the originally mentioned bugs. In the Table 6 & 7, we can see that the value for some projects scenarios is "N/A", indicating that the projects could not be run owing to memory limitations in the free SonarQube version.

Analysis by SpotBugs The tables 3, 4, show how well the SpotBugs analyzer detected SStuBs of particular types. Spotbugs were supposed to detect two types of SStuBs, which corresponded to the original dataset's *"Missing Throws Exception"* and *"Delete Throws Exception"*. To make things easier, we combined the bug type as *"add_delete_exception"* and counted the number of bugs found by the analyzer.

Table 3: Result of SpotBugs on Java projects

Project Name	Bugs type	No.of Bugs
Activi.activi.	add_delete_exception	3
apache.zookeeper.	add_delete_exception	2
dropqizard.dropwizard.	add_delete_exception	5
facebook.react-native.	add_delete_exception	0
gradel.gradel.	add_delete_exception	4
libgdx.libgdx	add_delete_exception	4
google.guava	add_delete_exception	0
greenrobot.guava	add_delete_exception	3
ctripcorp.apollo	add_delete_exception	1
loopj.android	add_delete_exception	2

Table 4: Result of SpotBugs on maven projects

Project Name	Bugs type	No.of Bugs
Activi.activi.	add_delete_exception	3
Alluxio.alluxio	add_delete_exception	1
antlr.antlr4	add_delete_exception	1
Bukkit.bukkit	add_delete_exception	2
eclipse.vert.x	add_delete_exception	2
gephi.gephi	add_delete_exception	2
square.otto	add_delete_exception	2
wildfly.wildfly	add_delete_exception	1
naver.pinpoint	add_delete_exception	2
checkstyle.checkstyle	add_delete_exception	0

For example, table 3 shows that SpotBugs detected 3, 5, and 4 bugs of the promised categories for java projects like *"Activi.activi"*, *"dropwizard.dropwizard"*, and *"libgdx.libgdx"* respectively. For maven projects, SpotBugs detected also significant number of bugs of selected categories. For example, SpotBugs discovered 1 and 2 bugs for the projects *"apache.hadoop"* and *"Bukkit.bukkit"*, respectively, but SpotBugs found none for the project *"antlr.antlr4"*. SpotBugs performed admirably in detecting SStuBs practically compared to the theoretically promised bugs present in the selected 100 java and 100 maven projects.

Table 5, shows the overall performance of the static analyzer tools in detecting three categories of SStuBs: *"More Specific If"*, *"Define and Throw Exception"*, and *"Null Pointer Exception"* in chosen 100 maven and java projects. From the Table 5, we can see that, there are discrepancies in the theoretical and practical claims of the analyzers in identifying SStuBs.

We discovered that SonarQube detected more SStuBs in both of the promised categories than the dataset's theoretical promised SStuBs. For the maven projects, SonarQube identified 1140 and 1357 bugs of *"Merge_If"* and *"Exception"* categories. On the other hand, for Java projects, it identified 942 and 1475 of the specified categories respectively. The theoretical and practical claims of SpotBugs, for discovering bugs in maven projects, are nearly identical. For java projects, conversely, SpotBugs outperformed in detecting SStuBs of the promised categories. PMD was unable to locate any SStuBs belonging to the promised categories.

Table 5: Number of identified bugs of selected categories by Analyzers

Analyzers	Promised	Practical
SonarQube	maven (291), Java (953)	maven (1140 + 1357), Java (942 + 1475)
SpotBugs	maven (116), Java (95)	maven (115), Java (130)
PMD	maven (291), Java (953)	maven (0), Java (0)

Findings of RQ1 The findings of the first research question are following:

- SonarQube detected much more SStuBs than the dataset. The reason for more bugs detection might be selection criteria of the projects between authors'original dataset and ours selected projects.
- Both SonarQube and SpotBugs identified some SStuBs out of the dataset as well as both the analyzers missed some SStuBs defined by the dataset.
- SpotBugs failed on some projects because of build failure issue.
- SonarQube encountered memory issue for large projects as we used unpaid version of the tool and this version of SonarQube failed to detect bugs for those porjects.
- PMD failed to detect any sort of relevant SStuBs from the dataset.

6.2 Analysis to RQ2

From the result analysis part of RQ1, we know that out of 16 templates mentioned in the dataset, 3 of them matched with the theoretical promised of the static analyzer tools. We filtered out the 3 templates and found out the remaining 13 templates which were missed by each of the three analyzers. After going through all the rules of java and maven projects for the analyzers, we found no identical rules related to the remaining 13 templates of stupid bugs. Table 8 shows the types of SStuBs totally missed by the analyzer tools. The number of bugs of these types present in the original dataset is also shown in the table.

Reasons for Missed Bugs: We personally evaluated and categorized some of the missed bugs to better understand why the great majority of bugs are not found by the studied bug analyzers. We looked at a random selection of some bugs that no bug detector has picked up on and searched for any issue reports linked with the bug. Next, we went through the list of bug patterns supported by the bug detectors to see if any of them could have identified the bug. If there was any bug detector that was related to the bug, such as by addressing a similar bug pattern, we tried out different versions of the buggy code to figure out what's going on.

What we found after analysis is that the vast majority of missed bugs are domain-specific issues unrelated to any of the tools' available patterns. These bugs are caused by errors in the implementation of application-specific algorithms, usually because the developer failed to handle a certain scenario. Furthermore, without

Table 6: Bug Information of Java Projects (SonarQube)

	Database		SonarQube Detected		Matched	
	Merge_If	Exception	Merge_If	Exception	Merge_If	Exception
Activi.activi	9	0	9	23	2	0
JetBrains.Kotlin	3	0	7	11	0	0
alibaba.canal	5	2	18	8	1	0
alibaba.storm	1	1	5	7	0	0
elastic.elasticsearch	10	5	16	25	0	0
ReactiveX.RxAndroid	1	0	8	12	0	0
JetBrains.Intellij	123	39	N/A	N/A	N/A	N/A
albaba.druid	5	3	40	55	2	0
albaba.dubbo	5	0	23	29	2	0
IMAX	0	0	2	4	0	0
PhilJay	1	0	4	7	0	0

Table 7: Bug Information of Maven Projects (SonarQube)

	Database		SonarQube Detected		Matched	
	Merge_If	Exception	Merge_If	Exception	Merge_If	Exception
Alibaba.fastjson	2	0	10	38	2	0
netty.netty	12	3	46	9	1	0
android.platform	13	18	N/A	N/A	N/A	N/A
apache.flink	5	5	17	36	0	0
apache.incubator	5	0	13	12	0	0
facebook.react	0	0	11	14	0	0
google.exoplayer	13	1	15	11	0	0
albaba.druid	5	3	40	55	2	0
libgdx.libgdx	6	0	8	14	1	0
spring.projects	3	0	5	7	0	1
zxing.zxing	3	0	4	5	0	0

domain knowledge, many bugs emerge in ways that are difficult to recognize as unintended. An example for this category is Chart-5 [3], when calling *ArrayList.add*, throws an *IndexOutOfBoundsException*. This bug might have been spotted by the existing analyzer tool SpotBugs for out-of-bounds accesses to arrays, but it does not consider *ArrayList*, hence the bug remained unreported.

Table 8: Categories of Bugs missed by Analyzer tools

Pattern Name	Maven 100	Java 1000
Change Identifier Used	3265	22668
Change Numeric Literal	1137	5447
Change Boolean Literal	3169	1842
Change Modifier	1852	5010
Wrong Function Name	1486	10179
Same Function More Args	758	5100
Same Function Less Args	179	1588
Same Func. Change Caller	187	1504
Same Function Swap Args	127	612
Change Binary operator	275	2241
Change BUnary operator	170	1016
Change Operand	120	807
Less Specific If	215	2813

6.3 Analysis to RQ3

Our motivation behind the third research question was to find out whether we could improve the performance of one of the analyzer tools in detecting SStuBs. As we found no guidelines and information to improve the analyzer tools, we opted for the manual implementation of a simple analyzer to find two patterns of SStuBs: *"Same Function More Arguments"* and *"Same Function Less Arguments"*.

We created a scanner in Python that can read a single java file and identify the types of errors that we are trying to show improvement while capturing the errors while analyzing. The scanner output additionally includes the bug's line number as well as the function's anticipated number of parameters. We ran and tested the script on 20 simple java files. We manually entered the identified categories of bugs into the code file and ran the script against it to see if the bugs were detected or not.The 20 sample java projects, both bug-free version and with manually added bugs version can be found in this GitHub repo.

Table 9 shows the results of the manual analyzer in detecting the selected categories of bugs. We presented the results of 10 basic Java projects out of 20 for convenience. The project names, the number of manually added bugs, and the total number of bugs found by the manual scanner are all visible in the table. For example, we

Table 9: Manual Scan Results

Project Name	Bugs Added	Bugs Detected	Line Number	Exceptional
ArrayExamples	3	3	14,15,16	-
Average	2	2	14,34	-
BSort	3	3	47,49	-
EightQueens	5	5	37,48,49,66,177	-
EnhancedFor	4	4	6,13,18,19	-
FiletExample	3	4	25,28,45,55	45
InListVer3	2	3	53,54,76	76
MSort	3	3	36,38,39	-
WordCount	3	4	36,74,87,88	74
Life	2	3,	47,48	55

manually added 3 and 5 bugs to the projects *"ArrayExamples"* and *"EigthQueens"* respectively, and the analyzer discovered all of the added bugs and output the line numbers of the bugs found. On the other hand, for the projects, *"InListVer3"* and *"Life"*, the analyzer not only found the created bugs but also detected similar categories of bugs that was previously there.

```
public static void solveAllNQueens(char[][] board, int
     col, ArrayList<char[][]> solutions){
---------
}

44. public static ArrayList<char[][]> getAllNQueens(int
     size){
45. ArrayList<char[][]> solutions = new
     ArrayList<char[][]>();
46. char[][] board = blankBoard(size);
47. solveAllNQueens(board, 0, solutions);
48. solveAllNQueens(board, 0, solutions,0);
49. solveAllNQueens(board, 0);
50. return solutions;
51. }
```

The analyzer also displays the function's line number, as well as the incorrect and original number of arguments. For example, in the *"EightQueens"* project, the *solveAllNQueens* function can take 3 arguments. In lines 48 and 49 of the code files, we manually added two bugs, one with more arguments and the other with less arguments. The manual analyzer correctly identified those two bugs, including their line number and the expected number of arguments in the function.

Findings of RQ3: As the manual analyzer successfully detected the certain categories of bugs it was based on, if the script can be incorporated with any of the analyzer tools, the performance of the tool must be improved in a certain percentage.

7 THREATS TO VALIDITY

While investigating, at different steps we faced different challenges working with the analyzers or the projects that we downloaded. Thus, by the end of the project, the observations faced a few threats as there are with any empirical investigations. The selected analyzer tools, which may or may not be representative of a wider population, is one constraint. To minimize this threat, we chose three commonly used static analyzer tools in industry for detecting bugs and those we believe are representative of the current state-of-the-art.

For this research scope, we selected only 100 Maven and 100 java projects, which is another constrain. We limited the number of projects to minimize memory and runtime issues. Moreover, we used a free version of the analyzer tools which faced built issues and memory issues for large projects. However, the bugs were gathered independently of our work and had been used in previous bug-related studies. Despite our efforts, we cannot claim that our findings are generalizable beyond the objects analyzed.

Another threat to validity in our methodology for detecting bugs is that it has the potential to miss certain detected bugs and misclassify coincidental matches as detected bugs. We manually searched and evaluated the bugs of the selected three categories. In addition, we manually collected and counted the number of bugs found in each category and the number of bugs found that matched with the dataset. To address this threat, both authors discussed every candidates found and validated the collected result.

The manual analyzer's non-optimized implementation is another internal threat to validity. Only one java file can be scanned at a time by the analyzer. As a result, it is not suitable for large projects with nested directories. To minimize the threat, we can combine the codes of numerous files into a single file and use the manual analyzer to find bugs in the areas we've chosen. This method is still good to proceed so far because the analyzer does not check the code semantics rather finds the specific bug pattern in the overall code file.

Many of the projects of the dataset have been upgraded to a newer version from the time since the dataset was first released. As a result, the number of stupid bugs in each of the categories might be changed. Our results might be different for analysis. There is no concrete reference point for comparing our results for both many of the newly detected bugs and some missed bugs of the selected categories and finally validating the result.

8 CONCLUSION AND FUTURE WORK

In this work, we examine the effectiveness of the static analyzer tool in bug detection strategies for finding SStuBs. The experimental results show that the effectiveness of the analyzer tools in detecting particularly 3 categories of bugs is obviously better than others. The other categories of bugs are missed due to domain specific issue and absence of theoretical promised rules of the analyzers. Trying to enhance analyzer tools is a tough endeavor because most analyzers are commercial products with little guidelines for improvement from outside sources. So, we manually implemented a scanner and tested on random java files for detecting two specific SStuBs patterns and the results prove that incorporating with analyzer tools can significantly improve the performance of the tools in detecting SStuBs.

In future, we will work to improve the analyzer for large projects with nested directories, as our manual scanner can only scan one java file at a time. In order to improve the analyzer tools, we will also try to detect other SStuBs patterns and try to incorporate with the analyzer tools. Furthermore, we plan to work on more projects to make the result generalized. Beyond these, our findings will assist future researchers in determining the efficiency of such tools, as well as working on further improvements of the tools in basic stupid bug identification.

REFERENCES

[1] Rafael-Michael Karampatsis and Charles Sutton. How often do single-statement bugs occur? the manysstubs4j dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 573–577, New York, NY, USA, 2020. Association for Computing Machinery.

[2] Balázs Mosolygó, Norbert Vándor, Gábor Antal, and Péter Hegedüs. On the rise and fall of simple stupid bugs: a life-cycle analysis of sstubs. *CoRR*, abs/2103.09604, 2021.

[3] Andrew Habib and Michael Pradel. How many of all bugs do we find? a study of static bug detectors. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 317–328. IEEE, 2018.

[4] Richard Bellairs. What is static analysis? static code analysis overview.

[5] Source code analysis tools.

[6] Jiayi Hua and Haoyu Wang. On the effectiveness of deep vulnerability detectors to simple stupid bug detection. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 530–534. IEEE, 2021.

[7] Rjust. rjust/defects4j: A database of real faults and an experimental infrastructure to enable controlled experiments in software engineering research.

[8] David A. Tomassi. Bugs in the wild: Examining the effectiveness of static analyzers at finding real-world bugs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 980–982, New York, NY, USA, 2018. Association for Computing Machinery.

[9] Ehsan Mashhadi and Hadi Hemmati. Applying codebert for automated program repair of java simple bugs. *arXiv preprint arXiv:2103.11626*, 2021.