Are the Scala Checks Effective? Evaluating Checks with Real-world Projects

Xin Zhang[1,3], Jiwei Yan[2,3,†], Baoquan Cui[1,3], Jun Yan[1,2,3], Jian Zhang[1,3]
[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
[2] Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences
[3] University of Chinese Academy of Sciences
Email: {zhangxin19, yanjw, cuibq, yanjun, zj}@ios.ac.cn

2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS) | 978-1-6654-5813-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/QRS54544.2021.00107

***Abstract*—Static analyzers can assist developers in detecting flaws and improving software quality. An analyzer often has numerous checkers, each of which implements a different checking rule. These checks can create a lot of warnings in real-world projects, putting a lot of pressure on programmers to examine them. Thus, it is critical to assess the effectiveness of these checkers before putting them to use. Typically, time-consuming questionnaires or human assessments of the warnings are employed to evaluate the checkers, which results in inefficiency when applied to real-world work.**

The significance and accuracy of checkers are the topics of this research, with the first reflecting the developers' attention to the checkers and the second reflecting the false-positive rate. We focus on Scala checkers in particular because, despite the popularity of the Scala programming language, there has been little study on them. We propose a method for tracking warnings in real-world projects and assessing the two features for each checker. We use 115 checks and six well-known Scala apps to demonstrate our approach. Based on the 191k warnings delivered by these checkers, the approach can identify 154k false positives, and it finds that only around 1/5 of the checks can benefit developers.

***Index Terms*—static analysis tools evaluation, Scala compiler plugin, automated evaluation, project evolution.**

I. Introduction

Scala is widely used in development because it can run directly on the Java virtual machine (JVM) and combines some language features such as object-oriented and functional programming [1]. Scala code's increased size necessitates the use of coding style checkers. [2], [3].

Scala developers have two options for implementing checkers: (1) they can implement their tools as compiler plugins and perform inspections on abstract syntax trees (ASTs). Alternatively, (2) they can use frameworks such as SOOT [4] to inspect JVM bytecodes after compilation. Mainstream Scala checkers choose the first option because they can detect bugs related to Scala language features [5], [6].

In most cases, developers may add many checkers to a tool, each of which inspects for one particular coding rule. When applying their tools to real-world projects, numerous warnings and high false-positive rates cause problems for both users and developers. It takes time for checker users to go through all of the warnings. Besides, there are substantial gaps in code structures between developers' benchmarks and real-world projects, and complete testing for each checker would take longer than checker development. Thus, both checker users and developers demand an efficient checker evaluation approach.

In particular, the significance and accuracy of a checker are the most concerned by both users and developers, while the first measures the user preference for the checker, and the second reflects the false-positive rate. Some methodologies opt to interview users for evaluating the first aspect [7]–[9]. And for the second aspect, researchers can physically inspect warnings to assess the readability and repairability, or they compute false-positive rates by traversing warnings and making judgments based on personal experience [10]–[12]. Additionally, some researchers use bug injection to study the rate of false-positive and false-negative [13].

These methods are primarily reliant on human experience or specific standards, and they may become unreliable if the developers or interviewees change. Besides, bug injection cannot accurately represent real-world bugs, making it difficult to assess the checkers' accuracy. If there is no effective approach, users may spend a large amount of time analyzing and selecting cost-effective checkers [14]. Although user feedbacks may help developers improve their checkers, obtaining feedbacks through questionnaires or interviews is time-consuming.

Besides, developing an automated method for assessing checkers on real-world projects is difficult. The number of fixed warnings and the time for repairing in the project evolution typically indicate the significance of a checker. But there are two difficulties in tracking the same warnings and identifying the fix.

- Warnings are jumbled together in the compiler's output with no standard structure, which makes it hard to filter out the targeted warnings automatically.
- Code migration is common in development, resulting in changes in the information of the same warning. If there is a code movement between two versions, the warning message will change, making exact string matching fails. Thus, the warning would be wrongly classified as fixed in the succeeding version, resulting in a bias in calculating the significance of the checker.

Calculating the false-positive rate is a commonly used

†Corresponding author

2693-9177/21/$31.00 ©2021 IEEE
DOI 10.1109/QRS54544.2021.00107

method of assessing accuracy. But two qualities constitute barriers for estimating it in real-world projects.

- A warning's output is plain text, making it impossible to categorize it only based on its message, especially when warning code spans many lines and contains complicated code structures.
- It is challenging to bring various developers' viewpoints together and offer persuasive and efficient reasons for flagging false positives.

In this paper, we propose an automated approach for evaluating Scala checkers. Our approach may be used by checker users to identify acceptable checkers or to assess more checkers on their projects with minimal human intervention. Furthermore, checker developers may enhance the checkers by including trends or modifying their rules to report more useful warnings.

We extract regular expressions from checker implementations and utilize regular expressions to automatically gather warnings from compiler outputs to handle the first barrier in judging the relevance. If a warning lasts just a few revisions, it reflects developers prefer to fix it, and its significance in this project is rising. Thus, we can estimate the significance of a checker by calculating the average number of lasting versions of the warnings. We utilize a two-step matching strategy to locate the same warnings across projects and limit the impact of code movement in the second challenge. We present patterns for each checker and automatically build repair patches based on these patterns to address challenges in assessing the accuracy. By checking if a patch passes compilation and then computing the false-positive rate of the relevant checker, we can identify whether a warning is a false alarm.

To show the benefits of our approach, we apply it against 115 Scala checkers on six Scala open-source projects from GitHub. These checkers are from two state-of-art Scala static analyzers. We assess the significance and accuracy of the checkers based on the warnings they throw.

We can include from the experiment results that: (1) Checker false-positive rates are high in real-world projects, as we can discover 154k false alarms out of a total of 191k alerts. And (2) there is minimal correlation between a checker's importance and accuracy, implying that users would not pay more attention to checkers that report more warnings. Furthermore, (3) in the six projects with high relevance and low false-positive rates, just 1/5 of the checkers are present.

In conclusion, the following contributions are made in this paper:

- We proposed an automated method for assessing checkers in open-source projects from two commonly concerning aspects: significance and accuracy.
- We propose patterns for automatically generating repair patches. These patterns can help developers improve their checkers and help users filter warnings before doing a human inspection.
- We conduct an empirical study on 115 Scala checkers in collaboration with six open-source projects. And the findings on 191k warnings can benefit both users and developers.

II. Background and Motivation

This section explains how checkers interact with the Scala compiler before showing an example of compiler output, which includes four warnings and three different output formats. Then we will give a motivating example to explain the necessitate of using an automated evaluation approach.

A. Framwork of Scala Compiler Plugins

Mainstream Scala checkers act as compiler plugins to adapt to language features. Unlike frameworks that rely on bytecodes for analysis, these checkers make static analysis using the Scala compiler. In this part, we will briefly introduce how these checkers interact with the Scala compiler.

Figure 1 depicts the workflow between the Scala compiler and checkers. The Scala compiler takes source code and converts it to JVM bytecode. The compiler turns source code to ASTs in the *parser phase*, and type inference information is added in the *refchecker phase*. Checkers can retrieve ASTs following any phase and report warnings. Thus, warnings from checkers are mixed in with the compiler output.

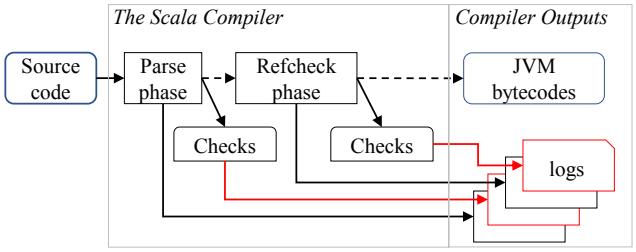

Fig. 1. The basic framework for Scala compiler plugins.

B. Form of Warnings

A warning contains two parts, where the Scala compiler provides the *position part*, and checkers define the *messsage part*. The file location of the warning, the line number of the warning code, and the associated source code are contained in the *position part*, whereas the *message part* comprises messages from checkers, i.e. a description for this warning or repair advice. As an example, we offer a piece of compiler output from project *Scala* [15] that has three sorts of warnings to demonstrate complicated warning forms.

Figure 2 includes four warnings in three different forms. The file path and line number appear first, followed by the message and warning code, and finally, a pointer pointing to the warning place. The first warning is from line 234 to line 236 and comprises a brief message in line 234, followed by warning codes in line 235. This notice warns users that erasing the abstract type will delete it.

The second warning are from line 279 to line 281, with the *message part* beginning on line 279 and containing the name of the related checker `YodaConditions` as well as a brief

```
1:   [info] scala-scala-53edc29\project
...
234: [warn] src\library\scala\collection\convert\
     WrapAsScala.scala:173: abstract type A in type pattern
      scala.collection.convert.Wrappers.
     ConcurrentMapWrapper[A,B] is unchecked since it is
     eliminated by erasure
235: [warn] case cmw: ConcurrentMapWrapper[A, B] => cmw.
     underlying
236: [warn]                              ^
...
279: [warn] src\library\scala\collection\immuTable\Vector.
     scala:108: [YodaConditions] Yoda conditions using you
     are.
280: [warn] if (0 < i) {
281: [warn]     ^
...
841: [warn] src\library\scala\Array.scala:193: [scapegoat]
      Variable shadowing
842: [warn] Variable shadowing is very useful, but can
     easily lead to nasty bugs in your code. Shadowed
     variables can be potentially confusing to other
     maintainers when the same name is adopted to have a
     new meaning in a nested scope.
843: [warn] val x: Float = _
844: [warn] for (x <- xs.iterator) { array(i) = x; i += 1
     }
845:            ^
...
5662: [warn] src\library\scala\collection\immuTable\
     RedBlackTree.scala:339: [scapegoat] Variable shadowing
5663: [warn] ...
5664: [warn] val left: scala.collection.immuTable.
     RedBlackTree.Tree[A,B] = _
5665: [warn] def unzipBoth(left: Tree[A, B],
5666: [warn]              ^
...
```

Fig. 2. Logs from project *scala*.

explanation. This warning indicates that the variable should be placed on the left side of the operator.

The third warning, which are from line 841 to line 845, cautions users that two variables have the same scope and name, which might lead to variable misuse. Line 842 includes the *message part*, which contains the tool name *scapegoat* and the checker name `Variable shadowing`. The warning code runs from line 843 to line 845, and it comprises two discontinuous codes from the source file. The fourth warning, which runs from line 5662 to line 5666, warns of the same problem as the third. The *position part* and the warning code are the only differences between the two warnings.

C. Motivating Examples

We will utilize two warnings in Figure 2 to demonstrate the limitations of state-of-art assessment methodologies when applied to real-world projects. The vast quantity of warnings, the difficulty in recognizing false positives, and the inability to identify whether a warning has been fixed are the three key constraints.

Existing methods rely heavily on manual inspection and hence cannot handle a high number of warnings. In our dataset, 115 checkers in project *elastic4s* [16] create over 126K warnings. The warning codes are extracted from roughly 21K Scala source code files, and the warning message is 462 Kloc in size, which is more than the project's size. It takes time for consumers to go through all of the warnings and assessing warnings at random would yield erroneous findings.

Recall the fourth warning in Figure 2, which informs users that the two variables in function `unzipBoth` have the same name and overlapping scope. The two warning variables are `left` at line 5665 and line 5664, with the first declared as a class member and the second specified as a function parameter in the source code. This warning alerts that variable shadowing is taking place, which might lead to variable misuse in the function.

To identify whether this warning is false-positive, the users must find the two cautionary variables in the source code. The first variable appears on line 435 of the source code, whereas the second appears on line 339. Next, the user must find all instances of the variable `left` in the function texttttunzipBoth and determine if variable shadowing would result in misuse in this function.

If code movement happens in the new version, knowing whether this warning is fixed is difficult because the warning's output has changed. In the previous version, the two warning variables appeared on lines 339 and 435 of the source file, but due to code migration, they now show on lines 325 and 379 in the current version. Besides, in the next version, the same checker reports a new warning and alerts two variables at lines 339 and 435. String matches will either not find the same warning in the next version, or treat the newly reported warning as the same unfixed warning. Because of the code relocation, simple string matching cannot establish whether the warning has been handled or is still present in the current version.

The limitations with present evaluating procedures in real-world software development encourage us to perform an empirical study of assessing checkers on real-world projects with less human labor and deliver useful conclusions for both users and developers.

III. Approach of Our Study

In this section, we present the details of the approach. Figure 3 depicts the structure, which consists of four modules. Regular expressions are required to extract warnings from

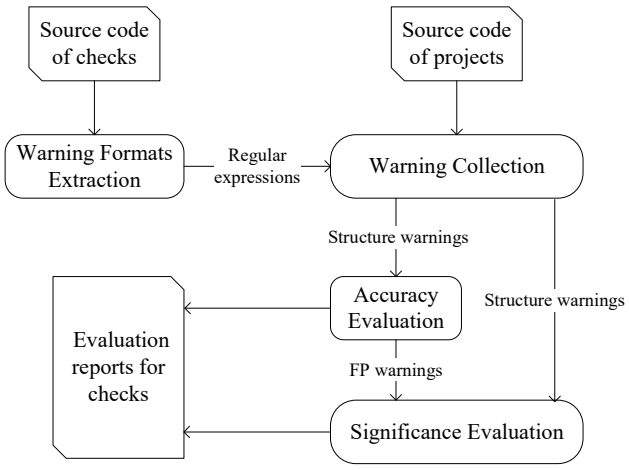

Fig. 3. Framework of the approach.

compiler output. So, in the *Warning Formats Extraction* module, we get regular expressions and pass them on to the next modules. We can partition one warning into the position and message parts after extracting warnings from compiler output in the textitWarning Collection module. A structural warning is a two-component warning that is sent to the following evaluation modules. In the following, we'll go through the specifics of each module.

A. Warning Formats Extraction

In this module, we extract regular expressions from the checkers' implementation to aid the future modules in evaluating these warnings.

As mentioned in Section II-B, a warning consists of two parts: the position and the message parts. We start by extracting regular expressions for both parts, then combining them to generate warning regular expressions. Because the Scala compiler defines the position component, the regular expression for this part is definite, which is `.*.scala:[0-9]*:,`. So all we have to do now is extract regular expressions for the message part from the checkers' implementation.

Figure 4 presents method `warn` in tool *Scapegoat* [17] which is used to format message part of a warning. In line 65, variable `report` contains the warning's message and it uses one of Scala language features, `String Interpolation`, in which `$text`, `$explanation` and `$snippetTest` are placeholders and will be replaced with their corresponding values during execution. We can conclude a regular expression from this interpolation string via replacing the placeholders with .*. After combined with regular expression from the position part, this module produces a regular expression *.*.scala:[0-9]*:[(a-zA-Z)*].*^* for checkers in this tool and passes it to the following modules.

```
30: def warn()
...
63: if(shouldPrint(warning)){
...
65:   val report = s"""|[scapegoat] $text |
    $explanation$snippetText""".stripMargin
...
```

Fig. 4. Source code from output module of a tool

B. Warning Collection

In the mainstream, Scala checkers are compiler extensions that can only provide warnings during compilation. We must first add these checkers into projects before compiling them to assess them. Because the Scala compiler can only load checkers from the same version, we must first set the checker's version to the same as the project's before loading it during compilation.

To automate this process and provide warnings, in this module, we first scan the project's configuration files for obtaining the Scala version and then add the corresponding checkers using the `ScalaOptions` command.

Figure 5 is part of a setting file *build.sbt* in project *Scala* [15]. We scan this setting file and get the Scala version it

```
53: val bootstrapScalaVersion = "2.11.5"
...
71: scalaVersion := bootstrapScalaVersion, // specify the
    Scala version
...
147: scalacOptions += "-Xplugin:scapegoat_2.11.jar", //
    inserted by the module
148: scalacOptions in Compile ++=
```

Fig. 5. A setting file in project *scala*

uses in line 53 and line 71, which is `Scala 2.11`. Then we find the use of command `ScalacOptions` in setting files, which is in line 148, and insert a checker in line 147.

After compiling the whole project, we use regular expressions to filter warnings from the compiler output. Regular expressions are also used to extract information from warnings and separate portions. A warning can be described by a 5-tuple: *[file path, line number, name of the check, description, warning code]*.

C. Accuracy Evaluation

Existing evaluation ways to categorize false-positive warnings involve manual inspection; however, because it is time-consuming and hard to harmonize the opinions of multiple participants, it is difficult to apply to a significant number of warnings. We manually propose patterns for each checker, and then we can automatically generate fixing patches based on the patterns for each warning. We can identify false positives with minimum human effort among numerous warnings by applying patches to the ASTs and compiling the updated ASTs.

We produce AST-based patches for each warning that includes a repair recommendation, as specified in the message part, and then apply the changes to ASTs during compilation. If the new ASTs compile successfully, we checker the output to see if the warning has vanished. If a warning fails to satisfy this test, it is considered a false-positive instance.

Figure 6 shows a warning with repair recommendation, which suggests replacing `Map.get.getOrElse` with `Map.getOrElse` in line 3. In Scala, API `Map.getOrElse(key, value)` would firstly examine whether the map object contains the `key`, and then returns the value in the map if it contains, or returns `value` if it does not contain.

```
1: [filepath]: ActorRefProvider.scala
2: [line number]: 620
3: [check's name]: Use of Map.get.getOrElse instead of Map.
    getOrElse
4: [description]: Map.get(key).getOrElse(value) can be
    replaced with Map.getOrElse(key, value), which is more
    concise.
5: [warning code]: ...
```

Fig. 6. One warning with repair recommendation.

Figure 7 shows a patch that we generate for this warning, in which the first line of each node is the node type and the second line is the corresponding source code. We first locate the node of `get(key).getOrElse(value)` on ASTs, which is the root node in the left ASTs in Figure 7,

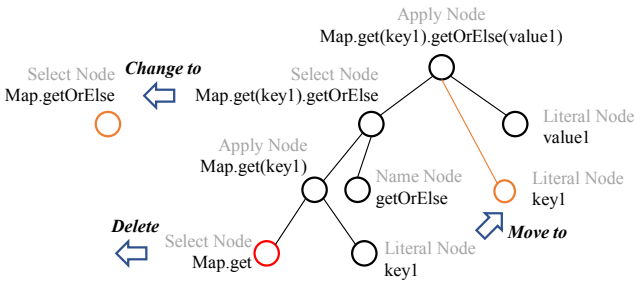

Fig. 7. Example for generating an AST-based patch.

and then store the two nodes of `key` and `value` from two `Literal nodes`. We then create a new `Apply node` corresponding to `Map.getOrElse(key1, value1)`, and fill two parameters with the two nodes stored previously, which is the right AST in Figure 7. We return the new ASTs to the Scala compiler after replacing the left ASTs with the right ASTs. We categorize this warning as false-positive if the modified ASTs could not pass compilation.

Patches are hard to be produced for certain checkers since there are no recommendations in the warnings. As a result, we choose to offer certain code context that the checkers must avoid to detect false positives. There is a standard code context for *Variable Shadowing* that will not be misused. The scopes of two variables do not overlap in Scala when a class member and a function argument have the same name. As a result, we may propose a code context for this check's warnings: if the two warned variables are a class member and a function argument, the warning is false-positive.

The third and fourth warnings in Figure 2 are from this check. For the third warning, we firstly separate the two warned variable `x` from line 843 and 844, and then locate `x` in line 843 under a `ClassDef Node`, and locate `x` in line 844 under a `LabelDef Node`. We do not consider this warning to be false-positive because the two nodes cannot fit the proposed context. For the fourth warning, we locate `left` in line 5664 under a `ClassDef Node`, and locate `left` in line 5665 under a `DefDef Node`. Because the proposed context can be found in both nodes, we consider this alert as false-positive.

D. Significance Evaluation

To get lasting versions for a warning, users must manually track warning changes in existing assessment methodologies, this becomes more time-intensive when the warning message is changed due to code migration.

In this section, we use a two-step matching strategy to automatically track warning changes and reduce the effect of the code migration. As shown in Section III-B, we can use a 5-tuple to describe a single warning. We match 5-tuples from two versions exactly in the first phase and consider the matched 5-tuples as if they are a single warning from both versions. For the remaining mismatched 5-tuples from the first step, which may entail code movement inside files, we ignore the line number in the 5-tuple for matching in the second step. After finding the same warnings in different versions, we can count

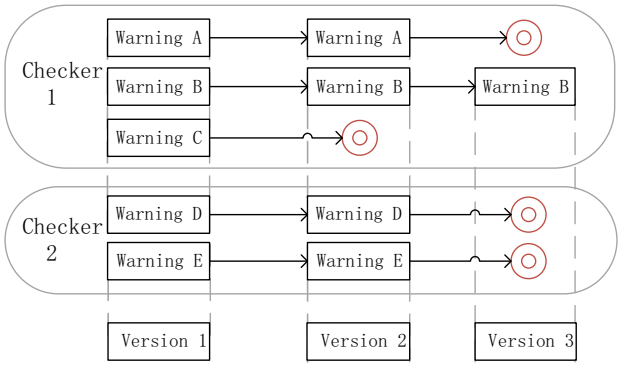

Fig. 8. Example for evaluating significance of checkers.

the number of lasting versions and use the average number to determine the significance of checkers. We don't know if the warnings that are still there in the most recent versions of our dataset have been fixed, so we leave them out.

Figure 8 shows two checkers with 5 warnings. In *checker 1*, *warning A* exists in *version 1* and *version 2*, and is fixed in *version 3*, while *warning B* exists in all three versions. We can evaluate the significance of *checker 1* with the average number of lasting versions of 1.5, and *checker 2* with 2. The results mean that warnings from *checker 1* are more likely to be focused by users than from *checker 2*.

We can alter cross-file code movement or code update by matching other sections of the 5-tuple, but we don't know if this modification is for warning correction or something else. As a result, we will not include these changes in this module.

IV. Study Setup

We present basic information about the checkers and projects in our dataset in this part, followed by the warning statistics. The data and findings in this paper are open to the public [18].

We select 115 checkers that can throw at least one warning on these projects from two widely-used Scala static analysis tools: *linter* [19] and *Scapegoat*, in which 51 checkers come from *Scapegoat* and 64 checkers come from *linter*. These checkers look for abuse of Scala APIs or duplicate code and they work with code written in Scala 2.10 to Scala 2.12.

To assess these checkers on real-world projects, we collect the top 50 Scala projects with the most stars on GitHub and then choose six projects that have released over 50 versions

TABLE I
Basic information of projects and warnings.

Projects				Warnings	
Name	LoC pv.[1]	Stars	Ver.[2]	N.W.10K.[3]	N.W.V[4]
scala	12950	435K	51	2237	3
akka	11147	24K	139	3335	38
gitbucket	8171	15K	213	331	25
scalaz	4392	58K	10	55	67
slick	2396	32K	113	368	20
elastic4s	1516	25K	103	259	150

[1] number of Scala source code per version.
[2] number of versions that we can successfully compile.
[3] number of warnings per 10 kLoC.
[4] number of warnings per version.

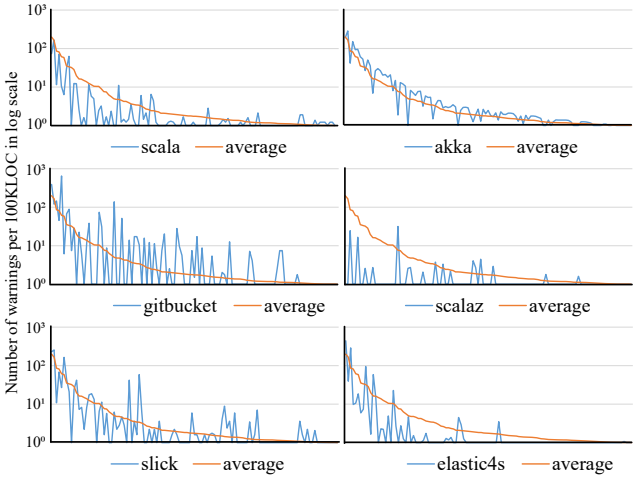

Fig. 9. Distribution of warnings for each project.

and are managed by sbt [20]. The basic information about the six projects is presented in Table I. The first column contains the project's name. The second column lists the number of lines of code for each release version, while the third column lists the project's stars. The projects in our dataset have above 50 release versions, as previously indicated. We cannot construct versions as some old dependencies are unavailable, and we only looked at versions that supported Scala 2.10 to Scala 2.12. We list the number of versions we can successfully create and gather warnings in the fourth column.

The 115 checkers are embedded in the six projects, and the compiler output yields almost 191K warnings. In Table I, the fifth and sixth columns show the number of warnings per 10 Kloc and version.

In Figure 9, we present the number of warnings thrown by each checker, with the checkers arranged by the number of warnings thrown per 100 kLoC. As can be observed, the number of warnings sent for each checker varies greatly between projects.

To study the universality of checkers, we compute how many projects a checker can spot warnings in and display the results in Table II. The first line of the table lists the number of projects on which a checker can issue warnings, and the second line lists the number of checkers that can report warnings in the same number of projects. As the data show, checkers perform differently in various initiatives. Most warning-generating checkers investigate using universal principles, whereas checkers that can only issue warnings on a few projects scrutinize using language features.

According to the statistical findings, the dataset utilized in our empirical study comprises projects with a variety of coding styles and checkers for both general and particular checking criteria.

TABLE II
NUMBER OF PROJECTS THAT A CHECK CAN THROW WARNINGS IN.

Number of projects	6	5	4	3	2	1
Number of checks	4	13	16	22	22	38

V. STUDY RESULTS

We propose three research questions in this section, which are

- **RQ1. What is the accuracy of checkers?** We will present the false-positive rates of checkers, and discuss the causes of the high false-positive rates of some checkers in RQ1.
- **RQ2. What is the significance of checkers?** In RQ2, we explore the significant distributions of checkers over all the warnings and see how false positives affect significance.
- **RQ3. What is the performance of our evaluation modules?** We will first discuss the efficiency of the warning tracking strategy, and then present the running time of the accuracy evaluation module in RQ3.

and by answering these questions we present the results of the empirical study.

A. RQ1. What is the Accuracy of Checkers?

The accuracy evaluation module's goal is to identify false-positive warnings. As described in Section III-C, we utilize two techniques to filter warnings that could not pass compilation or match patterns. We evaluate the accuracy of 53 checkers, using AST-based patches for 24 checkers and proposing patterns for 29 checkers, although our dataset contains 115 checkers. Some checkers do not provide repair recommendations, and the checking rules do not have special cases, so we evaluate the accuracy of 53 checkers.

To analyze the accuracy of the 53 checkers, and explore if one checker has similar accuracy across all projects, we first present the checkers' overall false-positive rates, followed by rates for each project separately.

In Figure 10, where the checkers are sorted according to false-positive rates, we compare the checkers' false-positive rates with the number of warnings they generate. The number of warnings is not related to the false positive rate, as shown, and half of the checkers have a false-positive rate of more than 40%.

For each check, we compare the rate on the six projects with the rate on one project, and the results are shown in Figure 11, where the checkers are similarly sorted based on the false-positive rate on the six projects. The data show that the false-positive rates of checkers vary greatly between projects, with particular checkers having high rates on more than half of them.

To analyze the cause of false positives, we supply one checker with the highest false-positive rate and another with

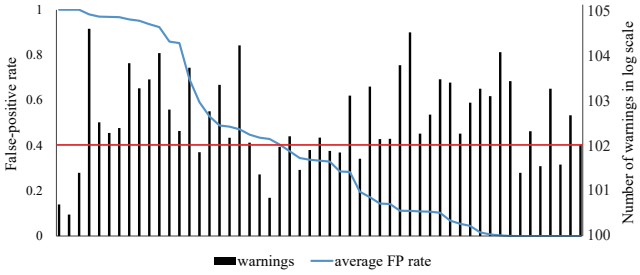

Fig. 10. False-positive rates and the number of warnings.

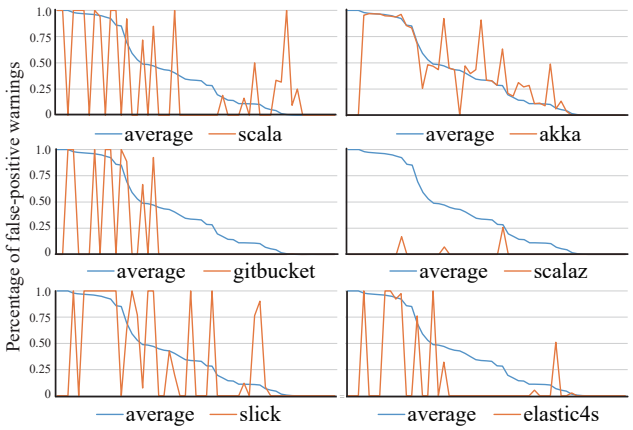

Fig. 11. False-positive rates in each project.

the highest number of warnings for each method, with the two checkers indicating the efficacy of the strategy in some way. Checker *While true loop* has the greatest false-positive rate, and checker *Variable shadowing* gives the most warnings (37715), with 97 percent being false positives. Checker *Lonely sealed trait* has the greatest false positive rate (86%) among the 24 checkers used to build AST-based fixes, while checker *UnusedParameter* gives the most warnings.

When the loop condition of a `while` loop is always true, checker *While true loop* issues a warning and indicates that an always true condition may lead to a dead loop. However, in some cases, a loop with a `true` loop condition will act as a polling function and will not collapse into a dead loop if it contains a `break` or `return` statement. So we check to see if the warning loop includes a `break` or `return` statement.

checker *Variable shadowing* inspects two variables that have the same name and nested scopes. As mentioned in Section III-C, we examine whether the two warned variables are a class member and function parameter.

If a class contains the modifier `sealed`, it cannot have any new subclasses unless the subclass is in the same file, according to the Scala grammars. Checker *Lonely sealed trait* examines if a sealed trait is inherited in the same file and warns about traits that are not. To do this check, we first identify the cautioned trait on the AST node, and then rename the trait by attaching the trait's hash value to the trait name; if this trait is used in the file, the compiler will raise errors if the trait name is altered. The characteristic is used in pattern matches in the same file, which leads to the majority of false-positive warnings in this checker.

The checker *UnusedParameter* looks for function parameters that aren't used in the function body. We identify the warned parameter on the AST node and then delete the node from the AST for the warnings in this check, since if this parameter is not utilized, it may be deleted with no influence on the compilation. We discover that 47 percent of warnings are false-positive, with the unused argument acting as a selector in overloaded routines. If the unnecessary argument is deleted, the compiler will generate an error indicating that the function has already been declared.

Based on the data, we can conclude that some fundamental code architectures may result in a large number of false positives from these checks, three of which check for duplicated code. By including code context in the inspection, developers may reduce false-positive alerts.

Conclusion. Based on the data, we can conclude that checkers have a considerable false-positive rate, particularly those with a high number of warnings. Furthermore, the false-positive rates of a single checker may vary greatly between projects. If the code structures from patterns and patches are often used in their projects, **users** can disable different checkers depending on the results without running the checkers. They can also deactivate various checkers on most projects with a high false-positive rate, as well as certain checkers with a 100% false-positive rate. **Developers** can utilize the patterns and patches to enhance checkers and include certain code structures. Developers may also improve their testing by including code structures into their benchmarks.

B. RQ2. What is the Significance of Checkers

The significance module makes an estimate based on the checkers' average lasting versions to determine their significance. If a checker has fewer versions with warnings, the module assigns it a higher priority in this project.

To see if the average number is appropriate for evaluating the significance and to estimate the impact of extreme numbers on the average value, we calculate the standard deviation, average, and the median number of warnings' lasting versions for each check, then display the three numbers in Figure 12, where we rank the checkers according to the ave.

As it shows, as the average value climbs, the median value fluctuates dramatically. Because the number of lasting versions varies greatly amongst checkers, as seen by the standard deviation, the median would be wrong when the number of warnings is less than 5 and the number of lasting versions varies greatly. When we examine checkers with a high standard deviation, this is a common occurrence in our dataset. The median and average values for checkers who have received many warnings are comparable. As a consequence, we base our estimation on the average number of long-lasting versions.

We provide the lasting versions in each project and compare them to the checkers' average lasting versions throughout the six projects to examine if a checker's significance is equivalent across all of them. The project name line and the median line in Figure 13 demonstrate the relationship between the checkers' average and the median number of lasting versions

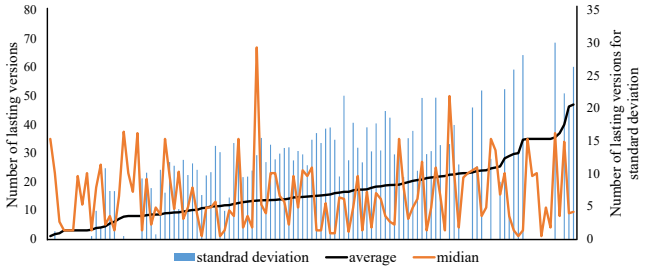

Fig. 12. The average, standard, and median number.

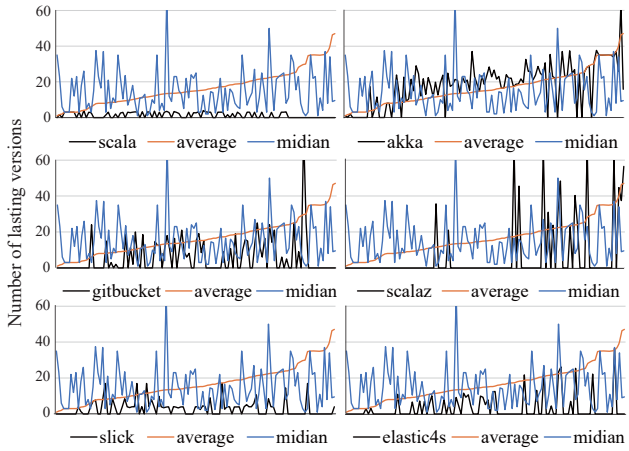

Fig. 13. Number of lasting versions in each project.

TABLE III
REDUCED NUMBER OF LASTING VERSIONS FOR EACH PROJECT.

Project	scala	akka	gitbucket	scalaz	slick	elastic4s
Average	0.39	7.00	7.09	6.81	6.09	5.73
Max	3	36	36	36	33	33
Median	0	5	4	4	3	3

in each project. The average line depicts the average number of the six projects. As can be observed, the long-term versions of checkers vary greatly amongst projects and have little relationship with the average value.

Besides, we compare the average lasting versions of checkers on all warnings and warnings without false-positive cases to explore the significance of checkers and estimate the influence of false positives. We illustrate the distributions of the reduced number of lasting versions in Figure 14. As can be observed, the number of lasting versions for most checks falls after filtering false-positive warnings. However, the number of false-positive warnings increases for 15 checkers after filtering, showing that these checkers have fewer false-positive warnings than others.

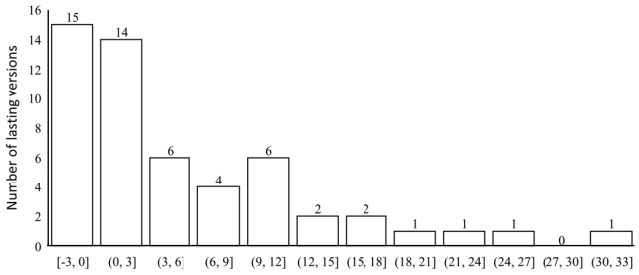

Fig. 14. Reduced number of lasting versions.

We explore the effect of false positives on checkers' lasting versions. After removing false positives from checkers, we compute the reduced number of lasting versions for each project and offer the average, maximum, and median figures in Table III. As indicated in the table, the false-positive alerts have a comparable effect on the significance.

Following the removal of false-positive warnings, we investigate the source of the increase in the number. We will use the checker *InefficientUseOfListSize* as an example because it has the highest number of lasting versions. This checker searches for the condition code that fetches the size of a list and then compares the number to other values, and it recommends instead using the API `isEmpty`. In some cases, the length of a list cannot be modified until it is compared to a value of 0 or 1, in which case we extract a pattern and provide it to the module. Once false-positive warnings are eliminated, the average number of enduring versions increases. We believe that project developers prefer to compare the length of a list to 0 or 1, rather than using `isEmpty` or other special values that would make the code difficult to understand, and that as a result, they are more likely to delete warning code that our method classifies as false positives.

We investigate the link between the accuracy and significance of one checker to see if a high false-positive rate leads to low significance. We sort the checkers twice for each project, once for accuracy and once for importance, with the accuracy ranking going from low false-positive rates to high rates and the significance ranking going from fewer enduring versions to more lasting versions. The rankings are open to the public on our website.

We compute the difference between the two ranks for each check. If the difference for one checker is closer to 0, it implies that high accuracy corresponds with high importance for that check. The distributions of difference for each project are shown in Figure 15. We can see that there is a weak relationship between accuracy and significance and that a high false-positive rate does not imply a poor significance.

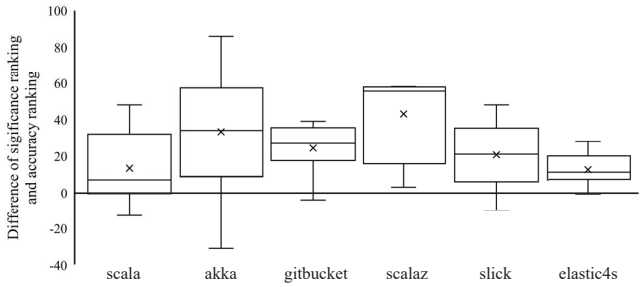

Fig. 15. Distributions of ranking difference

However, we can still find 1/5 of checkers with high accuracy and significance on the six projects, signaling that they can deliver more important warnings. These checkers are included in Table IV, which have a false-positive rate of less than 0.4 and a lifetime of fewer than 25 versions. We can see that just a few of these checkers seek API replacement, but the vast majority look for duplicate coding or irrational judgment.

Conclusion. The results show that the importance of inspections varies between projects and has a poor relationship with the number of warnings. Also, we can see that if a checker has low importance in one project, it is more likely to have low significance in other projects for more than half of the

TABLE IV
CHECKS WITH HIGH ACCURACY AND SIGNIFICANCE.

Check	acc[1]	sig[2]	Check	acc[1]	sig[2]
Avoid Traversable.size == 0	0	5	Avoid Traversable.size != 0	0	10
UseLastNotReverseHead	0	12	Use of Map.get().getOrElse	0	19
Broken odd check	0	20	Constant if expression	0	24
Use of Try.get	0.0007	17	Use of Option.get	0.002	14
Empty catch block	0.007	19	Unnecessary conversion	0.016	9
Incorrectly named exceptions	0.045	18	UseExistsNotFilterIsEmpty	0.054	22
Empty interpolated string	0.068	13	TypeToType	0.102	20
MergeNestedIfs	0.107	17	IdenticalCaseBodies	0.109	20
Use of asInstanceOf	0.111	13	UseExistsNotFindIsDefined	0.141	14
Paraless methods returns unit	0.171	18	UseOptionGetOrElse	0.196	17
InvariantCondition	0.333	4	UseIfExpression	0.3375	8
UseUntilNotToMinusOne	0.344	14	DuplicateIfBranches	0.375	16
Var could be val	0.404	8	UseLastOptionNotIf	0.428	7

[1] the false-positive rate.
[2] the average number of lasting versions.

115 checkers. If we filter the false positive warnings based on the accuracy module's results, these checkers can derive the lasting versions and raise the importance. Poor accuracy, on the other hand, does not imply low significance. Based on the findings, **users** could focus less on the checkers that aren't important for most projects and eliminate some checkers that have a high false-positive rate to speed up the warnings process. For **developers**, the findings of this module can help them understand how well the checkers function in real-world projects, as well as improve or remove checkers with low significance, particularly those that still have low significance after eliminating false positives.

C. RQ3. What is the Performance of Our Evaluation Modules?

This section will discuss how effective the tracking approach in the significance evaluation module is, as well as how long it takes to identify false-positive alarms.

As mentioned in Section III-D, we track warning changes by comparing the 5-tuples strictly and ignoring line number comparison to adapt to code movement inside a file. We compare the results of 5-tuple precisely matching with the results of our matching technique to see if this strategy is effective. We compare the two matching techniques for every two successive versions and see how many additional warnings our strategy can uncover. We divide the number of new warnings by the overall number of warnings, and the results are presented in Figure 16 for each project. As it shows, owing to code movement, half of the warnings would be missed in the precise match in most cases. Furthermore, we compare inside-file code movement to cross-file code movement by disregarding file path in matching, and we discover that cross-file code movement does not affect the warning track.

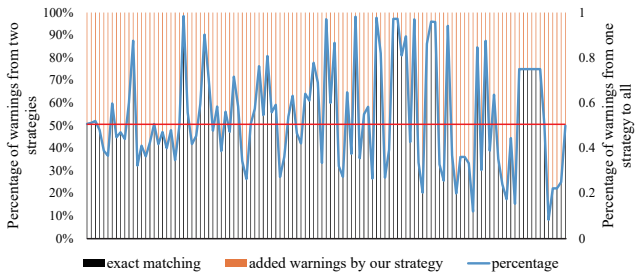

Fig. 16. Percentage of warnings under code movement.

TABLE V
TIME OF EXECUTION.

Project	scala	elastic4s	akka	slick	scalaz	gitbucket
Average	3.9	2.2	6.5	14.4	6.3	5.1
Max	157.7	43.5	126.6	190.5	198.3	17.8
Median	4.1	1.7	7.2	31.6	2.1	7.8

In the accuracy evaluation module, we develop a plugin to implement the two methodologies. Before deciding on a classification approach for each warning, the plugin first looks for the warning code on AST nodes. We use sbt's continuous compilation [21], and for each warning, we change the source code file's content to let sbt automatically recompile the file, allowing our plugin to load and receive ASTs for the amended file. As a result, we keep track of how long it takes to go from modifying a file to finishing recompilation.

We classify 191K warnings in 5253 milliseconds on average using a PC with an Intel Core E7-8850 and 1024GB RAM. Table V displays the average, median, and maximum execution times for each project. We can see that the maximum time is less than 200 seconds and that the majority of projects take less than 20 seconds. When compared to the time necessary to complete the project, we think the execution time is appropriate.

Because there may be disagreements in the accuracy module's patterns and patches, we publicly display one false positive for each check on our website.

Conclusion. The strategies we employ in our investigation performed well in terms of execution time and can overcome the obstacles posed by code mobility.

VI. THREATS TO VALIDITY

In this section, we discuss the threats to validity including internal validity and external validity for our study.

Internal validity. For our study, we only select 115 checks from two static analysis tools, and we are unable to evaluate the remaining checkers since they do not raise any warnings on the six projects. We believe that reviewing their conclusions is meaningless since they do not influence users because these checkers may be outmoded or the projects do not violate the regulations. Furthermore, some release versions have unknown dependencies and are difficult to build with plugins, resulting in a disparity between the number of long-lasting warnings and checks and the number in development. However, because we base our evaluation on a large number of versions and warnings, the results are more likely to reflect the efficacy of the checkers in real-world projects than other evaluation techniques.

Our warning tracking method is compatible with code movement; however, we cannot distinguish between code fix and code deletion, therefore we do not treat the code deletion as a warning fix. We believe that even an experienced developer will struggle to discern between the two instances and that we should focus on code movement rather than code deletion because the former has a greater effect on the significance.

We only use compilation to categorize false-positive rates for checkers, and because we couldn't cover all of the false-positive cases, we can only provide the lower bound of false-positive rates. However, the findings show that our strategies can filter more than 40% of warnings with less manual effort and execution time, potentially saving a significant amount of time for later warning analysis or checker assessment. For each check, users or developers must offer patterns or the right fixes. It is impossible to generate code automatically from a warning description, however, inspecting patterns and repairing patches is not difficult. As a result, we suggest that rather than automatically producing code, we should concentrate on pattern proposals.

External validity. Our findings are based on six Scala projects maintained with *sbt*; however, the results may alter if the projects are managed with other tools. On the other hand, the projects we choose have the most stars on GitHub and have been around for a long time, so they may demonstrate typical coding practices.

Because we only looked at 115 checkers, the results for other checks that rely on the Scala compiler plugin or JVM bytecode may be suspect. Although the strategies for identifying false-positive alerts have a significant influence on our dataset, they may not be appropriate for other applications. On the other side, the specific situations are rather common, and the repair patches are based on the description, which other checkers may utilize to improve their accuracy. The findings achieved by various third-party Scala compilers may differ since we based our research on a compiler plugin and acquired compilation results from the compiler.

VII. Related Work

In this paper, we produce an automated approach for evaluating checkers on the evolution of open-source projects. In this section, we summarize some related works on the warning analysis and analyzing tool evaluation.

Evaluation of static analysis tools. Existing studies evaluate the static analysis tools from a different perspective. Johnson et al. [7] interviews 20 developers to find out why compiler plugins are not used in development. Through some 40 to 60 minutes interviews, they conclude that unreadable output and high false-positive rate would be the barriers. Do et al. [22] implement a Just-in-Time plugin and embed it into development. They record some advice for improvement in using the tool. Tymchuk et al. [8] interview 14 full-experience developers to evaluate the feedback and adoption of plugins. Hovemeyer et al. [23] analyze the false positive case in FindBugs, and involve dataflow analysis to get more accurate results. They compare the warnings manually to explore whether dataflow analysis can find more defects with lower false-positive rates. Shen et al. [11] use a two-stage error ranking strategy for warnings thrown by FindBugs, and find the likelihood of bug patterns by reading the reports manually. Muske et al. [9] review 79 cases that handle defects and the approaches used in these cases. Mahmound et al. [24] compare several static analysis tools on Java and C/C++. They use four error types and test how many errors these tools could find out. Qiu et al. [25] compare static analysis tools that for taint analysis. They choose four tools and compare their results, and Figure out their performance on different kinds of sources and sinks. Delaitre et al. [26] let developers run 14 static analyzers on a specific benchmark, and then get reports from these developers. Using these reports they can compare these analyzers and find out which is the most welcomed.

Warning ranking. For static analysis tools that would throw plenty of warnings, there exist many studies that focus on warning ranking and recommendations. Shen et al. [11] present a tool for ranking warnings in FindBugs with the likelihood of bug patterns and bug kinds. Muske et al. [9] uses a survey to explore approaches that are used to handle warnings. HeckMan et al. [27] establishes a benchmark for tools, and classifies warnings from these tools. Kim et al. [28] propose a ranking algorithm that uses history warnings to mine warning fix experiences. Most studies that focus on warning ranking aim at using historical data or some features to rank the warnings.

Improvement of static analysis tools. To evaluate the compiler plugin, researchers have put forward many approaches to analyze and compare the tools. Bessey et al. [29] explain the importance of using compiler plugins in development and build tools to find common bugs in code. Layman et al. [30] conducts a study to learn what features developers care about when fixing errors, and conclude that the warnings need to be accurate and reliable, with precise descriptions. Ayewah et al. [31] conducts a study that observes users' performance when they use static analysis tools and conclude users' preference towards these static analysis tools. Johnson et al. [32] investigates some ways that current tools can improve, and conclude that tools need to have fast feedback and can be used in compilers or IDEs. Nagappan et al. [33] predicts the pre-release defects found by testing based on the defects found by static analysis tools, and can use defects thrown in testing to improve static analysis tools. Lewis et al. [34] uses a defect prediction algorithm in Google and finds that no changes in developer behavior. They conclude that tools need to meet the developers' orders. Mcallester et al. [35] use some approaches to improve static analyzers in the compiler. They focus on analyzers that are designed for optimization and test the soundness of these analyzers. Andreasen et al. [36] present some experience in improving the soundness and precision of a JavaScript analyzer. They involve delta debugging and blended analysis to help improve the analyzer.

VIII. Conclusion

In this paper, we describe an automated approach for analyzing the significance and accuracy of checkers on open-source project evolution, and we apply it to 115 Scala checks from six famous GitHub projects. According to the results of reviewing 191k warnings, there are 154k false positives among them, and for most checkers, more warnings or high accuracy would not result in high significance. On the six projects, 1/5 of checks had both high accuracy and significance, implying that they

can deliver more significant alarms. We provide repair patches and matching patterns for half of the checked checkers, which can help to explain the cause of false-positive alarms, and the methodologies we used in our study allow users to add more specific scenarios to the accuracy evaluation module. Our approach may help users choose checkers that conform to the coding styles of their projects, as well as developers identify the cause of false positives and improve code. In the future, other checks may be examined, and more projects may be included in the evaluation to cover more language features and coding styles.

ACKNOWLEDGEMENTS

This work is supported by the Key Research Program of Frontier Sciences, Chinese Academy of Sciences (Grant No. QYZDJ-SSW-JSC036), and the National 973 Program (Grant No. 2014CB340701).

REFERENCES

[1] Scala programming language. https://www.scala-lang.org.

[2] Karim Ali, Marianna Rapoport, Ondrej Lhoták, Julian Dolby, and Frank Tip. Constructing call graphs of scala programs. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, volume 8586 of *Lecture Notes in Computer Science*, pages 54–79, 2014.

[3] Yisen Xu, Xiangyang Jia, Fan Wu, Lingbo Li, and Jifeng Xuan. Automatically identifying calling-prone higher-order functions of scala programs to assist testers. *J. Comput. Sci. Technol.*, 35(6):1278–1294, 2020.

[4] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *Compiler Construction, 9th International Conference, CC 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*, volume 1781 of *Lecture Notes in Computer Science*, pages 18–34, 2000.

[5] Matic Potocnik, Uros Cibej, and Bostjan Slivnik. Linter: a tool for finding bugs and potential problems in scala code. In *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*, pages 1615–1616, 2014.

[6] Vicenç Torra. *Scala: From a Functional Programming Perspective - An Introduction to the Programming Language*, volume 9980 of *Lecture Notes in Computer Science*. 2016.

[7] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 672–681, 2013.

[8] Yuriy Tymchuk, Mohammad Ghafari, and Oscar Nierstrasz. JIT feedback: what experienced developers like about static analysis. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, pages 64–73, 2018.

[9] Tukaram Muske and Alexander Serebrenik. Survey of approaches for handling static analysis alarms. In *16th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2016, Raleigh, NC, USA, October 2-3, 2016*, pages 157–166, 2016.

[10] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, pages 470–481, 2016.

[11] Haihao Shen, Jianhong Fang, and Jianjun Zhao. Efindbugs: Effective error ranking for findbugs. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011*, pages 299–308, 2011.

[12] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 598–608, 2015.

[13] Asem Ghaleb and Karthik Pattabiraman. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, pages 415–427. ACM, 2020.

[14] Cristian Cadar and Alastair F. Donaldson. Analysing the program analyser. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, pages 765–768. ACM, 2016.

[15] Scala. https://github.com/scala/scala.

[16] Elastic4s. https://github.com/sksamuel/elastic4s.

[17] Scapegoat. https://github.com/sksamuel/scapegoat.

[18] Website for scala checker evaluation results. https://stardust1225.github.io/scala-check-evaluation/index.html.

[19] Linter. https://github.com/HairyFotr/linter.

[20] Sbt. https://www.scala-sbt.org.

[21] Sbt runtime. https://www.scala-sbt.org/1.x/docs/Running.html.

[22] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson R. Murphy-Hill. Just-in-time static analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 307–317, 2017.

[23] David Hovemeyer and William Pugh. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'07, San Diego, California, USA, June 13-14, 2007*, pages 9–14, 2007.

[24] Rahma Mahmood and Qusay H. Mahmoud. Evaluation of static analysis tools for finding vulnerabilities in java and C/C++ source code. *CoRR*, abs/1805.09040, 2018.

[25] Lina Qiu, Yingying Wang, and Julia Rubin. Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 176–186, 2018.

[26] Aurélien Delaitre, Bertrand Stivalet, Elizabeth Fong, and Vadim Okun. Evaluating bug finders - test and measurement of static code analyzers. In *1st IEEE/ACM International Workshop on Complex Faults and Failures in Large Software Systems, COUFLESS 2015, Florence, Italy, May 23, 2015*, pages 14–20, 2015.

[27] Sarah Smith Heckman and Laurie Williams. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *Proceedings of the Second International Symposium on Empirical Software Engineering and Measurement, ESEM 2008, October 9-10, 2008, Kaiserslautern, Germany*, pages 41–50, 2008.

[28] Sunghun Kim and Michael D. Ernst. Which warnings should I fix first? In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 45–54, 2007.

[29] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010.

[30] Lucas Layman, Laurie Williams, and Robert St. Amant. Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement, ESEM 2007, September 20-21, 2007, Madrid, Spain*, pages 176–185, 2007.

[31] Nathaniel Ayewah and William Pugh. A report on a survey and study of static analysis users. In *Proceedings of the 2008 Workshop on Defects in Large Software Systems, held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008), DEFECTS 2008, Seattle, Washington, USA, July 20, 2008*, pages 1–5, 2008.

[32] Brittany Johnson. A study on improving static analysis tools: Why are we not using them? In *34th International Conference on Software*

Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland, pages 1607–1609, 2012.

[33] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 580–586, 2005.

[34] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E. James Whitehead Jr. Does bug prediction support human developers? findings from a google case study. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 372–381, 2013.

[35] David A. McAllester. On the complexity analysis of static analyses. *J. ACM*, 49(4):512–537, 2002.

[36] Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. Systematic approaches for increasing soundness and precision of static analyzers. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2017, Barcelona, Spain, June 18, 2017*, pages 31–36, 2017.