RefDiff4Go: Detecting Refactorings in Go

Rodrigo Brito
ASERG Group - Department of Computer Science
Federal University of Minas Gerais (UFMG)
Belo Horizonte, Brazil
britorodrigo@dcc.ufmg.br

Marco Tulio Valente
ASERG Group - Department of Computer Science
Federal University of Minas Gerais (UFMG)
Belo Horizonte, Brazil
mtov@dcc.ufmg.br

ABSTRACT
Refactoring is a key software development practice that seeks to improve the internal structure of the code without changing its external behavior. In this way, the identification of refactorings is a key source of information for researchers, practitioners, and tool builders. However, existing approaches and tools documented in the literature do not support emerging ecosystems, such as the one of the Go programming language, which is nowadays widely used to develop robust and popular projects. To address these challenges, we present in this paper RefDiff4Go, a Go extension for RefDiff, which is a multi-language refactoring detection tool. RefDiff4Go detects 13 refactoring types in Go projects. We also evaluated RefDiff4Go with six well-known Go open-source projects, achieving 92% of precision and 80% of recall. Finally, we show that RefDiff4Go runtime performance supports its adoption in professional software projects.

CCS CONCEPTS
• Software and its engineering → Software maintenance tools.

KEYWORDS
Refactoring, Software Evolution, Software Repositories

ACM Reference Format:

1 INTRODUCTION
Refactoring is a key software development practice that seeks to improve the internal structure of the code without changing its external behavior [12]. Recent studies show that refactoring is often used in high-quality software systems and plays an important role in supporting maintenance and code evolution activities [22, 23].

In this way, the identification of refactorings is a key source of information for researchers intended to conduct empirical studies, providing relevant information about source code transformations. For example, refactoring data is crucial to understand key aspects of software evolution.

Recently, several studies have proposed automated approaches to detect refactorings activities [30, 32, 36, 37]. Tsantalis et al. [36] presented Refactoring Miner 2.0, a tool able of identifying 40 types of refactorings in Java systems with 99.6% of precision and 94% of recall. Other approaches, such as RefDiff, also present accurate detection techniques. The tool proposed by Silva et al. [39] has an extensible architecture, capable of detecting refactorings for Java, JavaScript, and C languages with 96.4% of precision and 80.4% of recall.

Currently, several studies focus on maintenance activities and the evolution of Java systems [7, 31, 34]. Tools such as RefDiff allow the expansion of such studies to other popular languages such as JavaScript and C, which are already supported. Also, RefDiff supports the addition of new programming languages, through a plugin system, permitting expansion to other relevant programming languages not explored in the literature, such as Kotlin, Go, and Typescript. Thus, there is a lack of tools that support developers and researchers to study refactorings in emergent programming languages. To address these challenges, we present RefDiff4Go, a RefDiff extension that detects 13 types of refactoring activities in Go projects. To our knowledge, we are the first to implement a detector for refactoring activities in Go. Particularly, our contributions are:

• An open-source RefDiff plugin for Go.
• A precision and recall evaluation with six well-known open-source projects.
• An oracle with over 68K performed refactorings.

The remainder of this paper is organized as follows: In Section 2, we present background information about extracting refactoring activities. Next, in Section 3 we present the proposed tool in detail. Then, in Section 4, we describe the design of the evaluation. Our results are reported in Section 5. In Section 6, we discussed the threats to validity. Finally, we present related works in Section 7 and concludes the paper in Section 8.

2 BACKGROUND

2.1 Go
Go is a statically typed and compiled programming language created by Google in 2007 to supply demands of large-scale and complex systems [25]. Currently, there are approximately 20K Go projects hosted on GitHub. Among these projects, there are popular and large software systems like Kubernetes (a container management system), Docker (a software containerization platform) and Terraform (a tool that manages and versions server configurations).

The syntax of Go is similar to C and C++, the language is statically typed and has only 25 keywords. Similar to C, Go uses structures
to define entities and functions. Besides that, the language has an abstraction mechanism through interfaces. However, Go does not have support for inheritance relations [11].

2.2 RefDiff

Recently, Silva et al. [30] proposed a technique and tool for detecting refactorings called RefDiff. The proposed approach consists of processing two revisions of a system through static analysis. The extraction process is divided into two main stages, Code Analysis and Relationship Analysis.

In the first step, RefDiff receives two revisions of the source code as input and builds a model called Code Structure Tree (CST), which is a tree structure where each node represents a code element, similar to an Abstract Syntax Tree (AST) but simpler. For example, CSTs provide information about the main elements of the code, such as functions, classes, and interfaces. This step is only applied to modified files. Typically, a CST is created from an AST. The selection of AST elements depends on the analyzed programming language and a set of rules defined by a RefDiff plugin. The plugin selects and converts each node and establishes relationships between them. For example, from Java ASTs, RefDiff extracts Classes, Interfaces, Enums, and Methods nodes. In the case of C, the plugin selects only Files and Functions. Each node type contains a unique identifier, and may receive the following custom attributes:

- **Name**: Identifier of the element in the code, usually its name. In the case of functions, the name is followed by the signature. For example, the `print` function in Figure 1 has the name `println(string)`.

- **Namespace**: An optional field, which in conjunction with the name field, identifies the global location of the node within the project. The namespace is associated only with high-level nodes like classes in Java and files in C. In the example shown in Figure 1, the namespace of the file is `main`, identified by package name.

- **Type**: A string that represents the type of the node in the analyzed language. For example, in Go the valid types are `struct`, `function`, `interface`, `type`, and `file`. In Figure 1, print has type `Function`.

- **Parameters List**: An optional field that identifies the names of the parameters in a function. In the example in Figure 1, the print function parameters is a list with a single element named value.

- **Tokenized source code**: This data is the source code of the node in the form of a list of tokens. For example, the list of tokens of the print function is: `func print(value string) { println(value) }`

![Figure 1: Example of Go file](image)

Besides, the CST structure is independent of the language structure, which allows integration with different programming languages. RefDiff extension is supported by a plugin system, a key component responsible for CST nodes identification, nodes relationship, and communication with RefDiff core. Currently, RefDiff supports three programming languages: Java, C, and JavaScript.

In the second stage, RefDiff compares the CST nodes between the two analyzed revisions. The ultimate goal of this stage is to discover relations between nodes that denote movements and modifications in the tree of objects.

The matching algorithm consists of two steps: search and classification. The first stage executes a recursive search in the node tree, analyzing attributes, such as signature, hierarchy, and similarity of the list of tokens. Subsequently, RefDiff follows a series of rules to classify the type of the relationship between each pair of nodes. For example, if two nodes have the same type, name, and parent they are connected by a `Same` relationship. Similarly, Refdiff applies a set of rules to identify each of the 13 supported refactorings. For a detailed specification of each of such rules, we refer the reader to RefDiff’s original paper [30].

Often, developers do not apply refactorings in isolation, i.e., commits can also fix bugs or include new features. Therefore, RefDiff also uses a similarity metric to identify possible variations in function bodies. This metric is computed using a Term Frequency-Inverse Document Frequency Algorithm (TF-IDF) algorithm [28], which weighs less frequent tokens with a higher score and reduces the relevance of common terms, such as brackets and semicolons.

<table>
<thead>
<tr>
<th>Refactoring</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same</td>
<td>Detection of identical elements in code</td>
</tr>
<tr>
<td>Convert Type</td>
<td>Transformation of definitions types</td>
</tr>
<tr>
<td>Change Signature</td>
<td>Changes function signature</td>
</tr>
<tr>
<td>Pull Up Method</td>
<td>Move a function to a superclass</td>
</tr>
<tr>
<td>Push Down Method</td>
<td>Move a function to subclasses</td>
</tr>
<tr>
<td>Rename</td>
<td>Renaming of code components</td>
</tr>
<tr>
<td>Move</td>
<td>Moving component location</td>
</tr>
<tr>
<td>Move and Rename</td>
<td>Move operation with renaming</td>
</tr>
<tr>
<td>Extract Function</td>
<td>Extraction of code to a new function</td>
</tr>
<tr>
<td>Extract Supertype</td>
<td>Extraction to a new shared superclass</td>
</tr>
<tr>
<td>Inline Function</td>
<td>Replace of a function call with its content</td>
</tr>
</tbody>
</table>

Table 1: Refactorings supported by RefDiff

At the end of the process, RefDiff returns a list of relationships in the form of a tuple \((n_1, n_2, Rel)\), where \(n_1\) is a CST node in the initial revision, \(n_2\) is a node in the compared revision, and \(Rel\) is the relationship identified between them. Table 1 lists the types of relationships, i.e., refactorings detected by RefDiff by default.

Figure 2 shows an example of relationships identified between two versions of a Go file. In Revision 1, we have one function named `avg`, which receives a list of values and return the average value. Next, we have the Revision 2 with two refactorings. The developer renames the function `avg` to `average` and extracts a new function from the original code with the summary logic.

In this example, the RefDiff returns a list with two tuples representing the relationship between revisions, \((avg, average, Rename)\)
which identify the rename refactoring, and \((\text{avg, sum, Extract})\) which represents the extraction of sum function.

## 3 REFDIFF4GO

![Figure 3: Refdiff4Go architecture](image)

RefDiff4Go identifies five main types of nodes: \(\text{Structs, Functions, Interfaces, Type Definitions (TypeDef),}\) and five different types of refactorings. Table 2 shows the relationship between the types of nodes identified and the possible refactorings. Table 2 shows the relationship between the types of nodes identified and the possible refactorings.

<table>
<thead>
<tr>
<th></th>
<th>Move</th>
<th>Rename</th>
<th>Extract</th>
<th>Inline</th>
<th>Change Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Struct</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>TypeDef</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Interface</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>File</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

![Table 2: Refactorings supported by RefDiff4Go](image)

Finally, the Call Graph module uses this information to create a graph of functions calls. Figure 4 shows an example of a Call Graph. The source code contains two function declarations: \(\text{format}\) and \(\text{run}\). The function format contains a single call to an external module. In this way, it includes a single edge to \(\text{Encrypt}\) node. The function \(\text{run}\), on the other hand, invokes two other functions, however, only \(\text{format}\) is included in the final graph, because \(\text{println}\) is a built-in method and therefore does not represent a possible CST node relationship. Figure 4 also shows the call graph generated for function \(\text{main.run}\), in the JSON format.

![Figure 4: Example of call graph representation](image)
shows an example of a `TypeDef` declaration, in which the user declared an array of integers under the name `mySlice`. Additionally, this new type also provides a function, which appends a new value in the array.

```go
define a regular expression to validate file extensions and suffixes (e.g. test.go, gen.go, pb.go). We also removed common utility folders such as test, testdata, and vendor (project dependencies).

Table 4 shows the number of refactorings detected in the analyzed projects. The project with most refactorings was Kubernetes, with approximately 52K refactorings. Besides that, the most frequent refactoring is `Change Signature`, with approximately 37K operations. By contrast, the lowest number of refactorings was detected in Gin, 203 refactoring activities, and the less frequent type is `Rename Type` with 57 refactorings.

Step 3. After creating a list of refactorings detected by RefDiff for Go, we randomly select ten samples for each kind of refactoring. It is also important to highlight that we selected at most one refactoring per commit (i.e., we never selected more than one refactoring per commit, to avoid possible bias due to preferences of a given developer or due to a major maintenance work under progress in the project).

Once the refactorings list was defined, the first author of this paper manually inspected the textual diff of each commit to check whether the identified refactoring was a True Positive (TP) or a False Positive (FP). A TP is a refactoring detected by RegDiff4Go that was indeed performed in the code, as confirmed in our manual validation. By contrast, a FP is a source code transformation that was not considered a refactoring in our manual validation.

Step 4. After completing the manual inspection, we compute precision using the formula:

\[
P = \frac{TP}{TP + FP}
\]

### 4.3 Computing Recall

To measure recall, we relied on the textual description of each commit of each project searching for possible documented refactorings. We also discarded commits with more than one parent in this step.

For example, to identify refactorings of type `Move`, we select commit messages with words such as move, moving, or migrate. The following example represents a commit message extracted from the Moby/Moby project, describing a `Move Struct` refactoring:

"Move Terms struct to os specific file"

However, some refactorings like `Inline Function` are not explicitly described in the commit message, so we decided to use terms with a closed related meaning, such as simplify or merge. The following commit message documents an `Inline Function` refactoring performed in one commit from Kubernetes/Kubernetes:

"Merge 3 resource allocation priority functions"

We finished our search after finding ten distinct commits for each refactoring operation supported by RefDiff4Go. In summary, our goal was to create a “ground truth” (or oracle) of refactoring operations. However, not all types of refactorings had documented changes, such as inline functions, from which we only found eight occurrences. In total, we collected 128 commits to compute recall.

However, it is also possible that a documented refactoring (in a commit description message) was not performed in the code. To discard such cases, the first author of this paper manually checked each

1https://github.com/moby/moby/commit/a70dd659
2https://github.com/kubernetes/kubernetes/commit/c65225ee
commit in our initial ground truth, searching for the documented refactorings in the respective commit diff. To replace these commits, we performed a new draw with a second manual inspection.

A False Negative (FN) is a refactoring from the ground truth that is not detected by RefDiff4Go. Finally, to compute the recall score, we use the following formula:

$$R = \frac{TP}{TP + FN} \quad (2)$$

We also combine the precision and recall using a harmonic mean, also known as F1-Score. We obtain the F1-Score value applying the following formula, where R is the recall value and P represents the precision:

$$F1 = 2 \cdot \frac{P \cdot R}{P + R} \quad (3)$$

Table 5 shows the total number of commits analyzed per refactoring type for the precision and recall evaluation. In total, we inspected 258 refactorings, 130 for precision and 128 for recall.

5 RESULTS

In this section, we present the results of the RefDiff4Go evaluation. Table 6 presents the results for precision, recall and F1-Score for each type of refactoring. Precision ranges from 0.70 (Move Function) to 1.0 (Inline Function, Move File, and Rename Struct). The average precision is 0.92. Recall ranges from 0.5 (Inline Function) to 1.0 (Rename Struct and Rename File), reporting an average of 0.80 therefore slightly lower than precision. Table 6 also shows the F1 scores for each refactoring operation. The average F1 score is 0.86.

As presented in Table 6, some refactorings like Inline Function reports a low recall value (0.67). In this way, we investigated some commits related to this class of refactoring searching for possible
causes. By definition, Inline Function replaces a function call by the function body. However, after the replacement, developers often modify the inlined code. Since RefDiff4Go has only access to the full modification performed in the commit (i.e., the tool works in a commit granularity level), it considers that the new code is very different from the original function body. As a result, the tool does not detect an Inline Function refactoring operation.

Next, we give a real example. Commit 6a42e1, from the Kubernetes project, received the FN classification for the Inline Function refactoring. The commit message documents the refactoring with the following message: “Inline/simplify two used-only-once service test helper functions.” As a result, we included the Inline Function refactoring showed in Figure 6 in our ground truth, where the call of function LaunchNetexecPodOnNode is replaced by its body.

However, as we can see, the code transformation was not a pure Inline Function, since the inlined function body was also modified. The manual validator considered that this modification does not mischaracterize the Inline Function refactoring. On the other side, this modification prevented the trigger of an Inline Function by RefDiff4Go. Thus, this example shows that there is a degree of subjectivity when refactoring operations are evaluated under the granularity of commits.

A similar case is observed in Commit 0e04b9 from Hugo project, where the developer has documented the following message: “Moving processing short codes to the page.” The operation performed by the developer consists of moving a function between two files. RefDiff4Go did not identify the refactoring activity and also received an FN classification. Similar to the Inline Function refactoring presented earlier, the developer moves the code, but also changes its content. Figure 7 presents the described operation.

---

3https://github.com/kubernetes/kubernetes/commit/6a42e1
4https://github.com/gohugoio/hugo/commit/0e04b9

---

Figure 6: Example of Inline Function from Kubernetes project
Therefore, the results showed that refactorings related to code movement — such as extract, inline, and move — are more susceptible to failures when the developer performs the operation and also modifies the behavior of the target code. However, the proposed tool can identify sub-operations contained in several cases with similar behavior, thus reducing the failure impact.

5.1 Comparison with Java, JavaScript and C

In this section, we present a comparison between RefDiff4Go and the other programming languages supported by RefDiff: Java, JavaScript and C. However, it is not possible to compare all refactoring types. For example, Pull Up Method and Push-Down Method refactorings are only detected in RefDiff for Java.

On the other side, some refactoring types have similarities, allowing a direct comparison. For example, a Struct component in Go behaves similarly to a Class in Java and JavaScript. Table 7 shows the selection of compatible refactorings for each language. As another important point to highlight, the results presented in this section are restricted to the refactoring types evaluated in a recent study by RefDiff [30].

<table>
<thead>
<tr>
<th>Refactoring</th>
<th>Go</th>
<th>Java</th>
<th>JavaScript</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Sign</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Change Sign</td>
</tr>
<tr>
<td>Extract Func</td>
<td>Extract Meth</td>
<td>Extract Func</td>
<td>Extract Func</td>
<td></td>
</tr>
<tr>
<td>Inline Func</td>
<td>Inline Meth</td>
<td>Inline Func</td>
<td>Inline Func</td>
<td></td>
</tr>
<tr>
<td>Move Struct</td>
<td>Move Class</td>
<td>Move Class</td>
<td>Move Class</td>
<td></td>
</tr>
<tr>
<td>Move Func</td>
<td>Move Meth</td>
<td>Move Func</td>
<td>Move Func</td>
<td></td>
</tr>
<tr>
<td>Rename Struct</td>
<td>Rename Class</td>
<td>Rename Class</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Rename File</td>
<td>-</td>
<td>Rename File</td>
<td>Rename File</td>
<td></td>
</tr>
<tr>
<td>Rename Func</td>
<td>Rename Meth</td>
<td>Rename Func</td>
<td>Rename Func</td>
<td></td>
</tr>
</tbody>
</table>

Table 7: Refactorings compatibility considered in the evaluation

5.2 Java

Table 8 shows the comparison between RefDiff4Go and the RefDiff plugin for Java, we present the F1 score for each refactoring and the number of refactorings used in precision and recall analysis. It is also important to observe that the number of refactorings used in the evaluation of the Java plugin is greater than Go. In total, we compared 3,023 refactorings from Java study with 118 refactorings in Go.

<table>
<thead>
<tr>
<th>Refactoring</th>
<th>Go F1-Score</th>
<th>#</th>
<th>Java F1-Score</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract Function / Method</td>
<td>0.90</td>
<td>20</td>
<td>0.78</td>
<td>1,037</td>
</tr>
<tr>
<td>Inline Function / Method</td>
<td>0.67</td>
<td>18</td>
<td>0.82</td>
<td>122</td>
</tr>
<tr>
<td>Move Class / Struct</td>
<td>0.85</td>
<td>20</td>
<td>0.98</td>
<td>1,100</td>
</tr>
<tr>
<td>Move Function / Method</td>
<td>0.75</td>
<td>20</td>
<td>0.84</td>
<td>319</td>
</tr>
<tr>
<td>Rename Class / Struct</td>
<td>1.00</td>
<td>20</td>
<td>0.90</td>
<td>95</td>
</tr>
<tr>
<td>Rename Function / Method</td>
<td>0.95</td>
<td>20</td>
<td>0.80</td>
<td>350</td>
</tr>
<tr>
<td>All</td>
<td>0.85</td>
<td>118</td>
<td>0.85</td>
<td>3,023</td>
</tr>
</tbody>
</table>

Table 8: Comparison of results between RefDiff4Go and Java

Considering the average F1-score, the results for Go and Java are exactly the same: F1-score is 0.85 for both languages. RefDiff4Go performed better than the Java plugin for Extract Function, Rename Struct and Rename Function refactorings.

Summary: RefDiff4Go presents a similar result to the Java plugin, recording the final F1-Score of 0.85, which is also reported by Java plugin.

5.3 JavaScript

We also compared the results obtained by RefDiff4Go with the JavaScript plugin. Table 9 shows the F1-Score for compatible refactorings. It is also worth mentioning in this case, the number of
refactorings used in the Go results is higher (158 refactorings) than the ones used for JavaScript (122 refactorings).

<table>
<thead>
<tr>
<th>Refactoring</th>
<th>Go</th>
<th>JavaScript</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F1-Score</td>
<td>#</td>
</tr>
<tr>
<td>Extract Function</td>
<td>0.90</td>
<td>20</td>
</tr>
<tr>
<td>Inline Function</td>
<td>0.67</td>
<td>18</td>
</tr>
<tr>
<td>Move Struct / Class</td>
<td>0.85</td>
<td>20</td>
</tr>
<tr>
<td>Move File</td>
<td>0.82</td>
<td>20</td>
</tr>
<tr>
<td>Move Function</td>
<td>0.75</td>
<td>20</td>
</tr>
<tr>
<td>Rename Struct / Class</td>
<td>1.00</td>
<td>20</td>
</tr>
<tr>
<td>Rename File</td>
<td>1.00</td>
<td>20</td>
</tr>
<tr>
<td>Rename Function</td>
<td>0.95</td>
<td>20</td>
</tr>
<tr>
<td>All</td>
<td>0.87</td>
<td>158</td>
</tr>
</tbody>
</table>

Table 9: Comparison of results between Refdiff4Go and JavaScript

JavaScript and Go plugins present a similar result. RefDiff4Go presented a lowest score only in Move Function (0.75) against 0.95 and Move File (0.82) when the JavaScript plugin presents 1.0 of F1 score. It is worth mentioning that we can not calculate the F1-Score for Move and Rename Struct/Class obtained by JavaScript, once the authors not reported the precision value.

Summary: RefDiff4Go presents a better result than JavaScript plugin, reporting a F1 score of 0.87 in comparison of 0.83 presented by JavaScript.

5.4 C

Table 10 presents the results of the comparison between RefDiff4Go and the C plugin. Differently from other languages, C and Go have similar types of refactorings. Thus, we did not need to rely on equivalences when comparing the results. However, although both languages have Struct as the principal component of abstraction, the C plugin does not support refactorings involving this kind of component, such as Move Struct and Rename Struct.

<table>
<thead>
<tr>
<th>Refactoring</th>
<th>Go</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F1-Score</td>
<td></td>
</tr>
<tr>
<td>Change Signature</td>
<td>0.85</td>
<td>20</td>
</tr>
<tr>
<td>Extract Function</td>
<td>0.90</td>
<td>20</td>
</tr>
<tr>
<td>Inline Function</td>
<td>0.67</td>
<td>18</td>
</tr>
<tr>
<td>Move File</td>
<td>0.82</td>
<td>20</td>
</tr>
<tr>
<td>Move Function</td>
<td>0.75</td>
<td>20</td>
</tr>
<tr>
<td>Rename File</td>
<td>1.00</td>
<td>20</td>
</tr>
<tr>
<td>Rename Function</td>
<td>0.95</td>
<td>20</td>
</tr>
<tr>
<td>All</td>
<td>0.85</td>
<td>138</td>
</tr>
</tbody>
</table>

Table 10: Comparison of results between Refdiff4Go and C

In general, both tools achieved similar results. For example, RefDiff4Go presented a better results for Extract Function (0.9), Inline Function (0.67), against 0.82 and 0.64 for C plugin, respectively. However, RefDiff4Go had a worst result mainly for Move File, reporting a F1-Score of 0.82 against 1.0 of C.

Summary: RefDiff C plugin presents a slightly better result, reporting a F1 score of 0.87 against 0.85 presented by Go.

5.5 Execution Time

Besides comparing recall and precision, we also evaluated the execution time spent by RefDiff4Go. For this purpose, we started with the 110,728 commits from our initial dataset. Then, we discarded commits with more than one parent (e.g. merge commits) and commits that did not contain changes in source code. The final dataset used to measure the execution time contains 70,645 commits and 68,204 refactorings.

Figure 8 presents a graph of the execution time by commit, using a logarithmic scale. To perform the tests, we used a Core I5-7400 computer, 16 GB of RAM running Linux operating system, Ubuntu 18.04 distribution.

We can notice that RefDiff4Go presented a median execution time of 171 milliseconds. However, we also identify a large number of commits (14.9%) that are processed in less than ten milliseconds, which are basically commits with trivial changes in small files. As another point to remark, a small fraction of the commits (1.7%) required more than 10 seconds to execute. Normally, these commits perform massive project changes. An example can be found in Kubernetes where more than 1,900 files were modified in a single commit. Figure 9 shows the execution time for each analyzed systems. It is possible to see that Gogs is the system with the longest execution time, with median of 406 milliseconds, followed by Hugo (median 278ms) and Moby (median of 246ms).

6 THREATS TO VALIDITY

Dataset: Github has thousands of Go projects and we have built our dataset using the top-6 code projects by number of stars. Therefore, our dataset represents a small sample of the entire universe of Go projects.

5https://github.com/kubernetes/kubernetes/commit/fb9bb501
Refactoring is a key practice to evolve and preserve software systems. Due to the importance of such activities, there is a large body of studies on the motivations for performing refactoring [20, 31, 35], on the challenges of refactoring [15, 16], impact on software quality and evolution [5, 9, 18], security [1, 4, 21] and, providing tools to support developers in evolution and maintenance activities [2, 3, 8, 13, 19, 29, 34].

For example, there are several studies in the literature based on the identification of refactoring activities. It is possible to find different approaches, such as plugins integrated to IDEs during software development [14, 24, 26], metadata analysis of revision control systems [17, 27], and static analysis [30, 32, 35].

The most common approach for detecting refactorings is static analysis. Currently, we find in the literature several studies that apply this technique. There are tools focused on detecting refactorings in a single programming language [10, 32, 36, 37] and approaches for identifying refactorings activities in multiple languages [30].

Dig et al. [10] proposed Refactoring Crawler, a static analysis tool that detects seven types of refactoring: Change Method Signature, Rename Package / Class / Method, Pull Up Method, Push Down Method, and Move Method. The tool is based on a lightweight static analysis using the Shingle encoding technique combined with a semantic analysis to refine the results. The approach proposed by the authors presented good results in a study with three relevant projects, EclipseUI, Struts, and JHotDraw, reporting accuracy of 85%.

Ref-Finder, proposed by Prete et al. [14, 26], is a tool that covers a wide range of refactorings. Through an Eclipse plugin, the tool identifies atomic and composite refactorings using a template-based refactoring reconstruction approach, covering 63 types of refactorings from Fowler’s catalog [12]. The authors evaluated the tool with examples from Fowler’s catalog and real open-source projects, reporting a precision of 0.79 and recall of 0.95.

Tsantalis et al. [35] proposed Refactoring Miner, a tool capable of identifying 14 types of refactoring in Java projects: Move Class / Method / Field, Extract Method, Inline Method, Rename Package / Class / Method, Pull Up Method / Field, Push Down Method / Field, and Extract Superclass / Interface. The approach proposed by the authors consists of a lightweight version of UMLDiff [38] for analyzing object-oriented models. The tool was applied in an empirical study in three well-known projects JUnit, HTTPCore, and HTTPClient. The results report an accuracy of 96.4% for Extract Method, 97.6% for Rename Class, and 100% accuracy for the other types of refactorings analyzed.

In 2018, Tsantalis et al. proposed a new tool called RMiner [37], an evolution of the approach previously presented. RMiner is based on an AST matching algorithm and supports 15 different types of refactorings. In a new study, the authors evaluated the tool involving real 3,188 refactorings from an oracle. In this large oracle, RMiner showed a precision of 98% and a recall of 87%.

In a recent study, published in 2020, Tsantalis et al. [36] presented Refactoring Miner 2.0, an evolution of the tool previously presented with improvements in the matching algorithm. The authors extended the tool to support 40 different types of refactoring. In a large-scale study, involving 7,226 real instances of refactoring, Refactoring Miner showed an average precision of 99.6% and recall of 94%, surpassing the result of six other compared approaches, both in accuracy and execution time.

Silva et al. [32] proposed a new approach for detecting refactorings activities called RefDiff 1.0. The tool uses a combination of heuristics through static analysis and TF-IDF as a measure of similarity. RefDiff is able to identify 13 types of refactorings. Besides, the tool was evaluated through an empirical study and compared with three known tools: Refactoring Miner, Refactoring Crawler and Ref-Finder. The tool proposed by the authors outperformed the others, recording 100% of precision and an 88% of recall.

Recently, Silva et al. presented a new version of RefDiff [30], now named RefDiff 2.0, and with multi-language support. The new approach is able to identify 11 different types of refactoring in three
programming languages: Java, JavaScript, and C. The proposed algorithm is based in a Code Structure Tree (CST) that abstracts the structural representation of the code and, therefore, is independent of the syntax of programming languages. RefDiff 2.0 also allows extension via plugins to support new languages. In the study, the authors reported a 96% accuracy and 80% recall.

8 CONCLUSION

In this paper, we presented RefDiff4Go, an extension of RefDiff for detecting refactorings in Go software projects, capable to identify 13 different types of refactoring. We evaluated the tool with six well-known open-source projects and compared them with plugins from other languages: Java, JavaScript, and C. The results report a precision of 92% and 80% of recall, resulting in an F1 Score of 0.86. The evaluation also showed that RefDiff4Go presented a similar accuracy than other language plugins and good execution time, recording a median of 171 milliseconds per commit.

As future work, we plan to execute a survey with developers to better understand their needs and include new promising languages to RefDiff. In addition, to looking for practical ways to apply RefDiff4Go on the fly to assist developers during maintenance and code review activities. The developed tool is publicly available on GitHub.6

9 ACKNOWLEDGMENTS

This research is supported by grants from CAPES and CNPq.

REFERENCES


6https://github.com/rodrigo-brito/refdiff-go


