Accurate Method and Variable Tracking in Commit History

Mehran Jodavi
Concordia University
Canada
m_jodavi@encs.concordia.ca

Nikolaos Tsantalis
Concordia University
Canada
nikolaos.tsantalis@concordia.ca

ABSTRACT

Tracking program elements in the commit history of a project is essential for supporting various software maintenance, comprehension and evolution tasks. Accuracy is of paramount importance for the adoption of program element tracking tools by developers and researchers. To this end, we propose CodeTracker, a refactoring-aware tool that can generate the commit change history for method and variable declarations with a very high accuracy. More specifically, CodeTracker has 99.9% precision and recall in method tracking, surpassing the previous state-of-the-art tool, CodeShovel, with a comparable execution time. CodeTracker is the first tool of its kind that can track the change history of variables with 99.7% precision and 99.8% recall. To evaluate its accuracy in variable tracking, we extended the oracle created by Grund et al. for the evaluation of CodeShovel, with the complete change history of all 1345 variables and parameters declared in the 200 methods comprising the Grund et al. oracle. We make our tool and extended oracle publicly available to enable the replication of our experiments and facilitate future research on program element tracking techniques.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; *Software configuration management and version control systems.*

KEYWORDS

commit change history, refactoring-aware source code tracking

ACM Reference Format:
Mehran Jodavi and Nikolaos Tsantalis. 2022. Accurate Method and Variable Tracking in Commit History. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549079

1 INTRODUCTION

Developers routinely track code snippets in the commit history to facilitate various software engineering tasks. Codoban et al. [6] surveyed 217 developers to find the motivations behind examining software history. The most common reasons are to a) recover the rationale behind a snippet of code, b) find the commits that introduced a bug, c) find who are the knowledgeable peers on certain modules and patterns, d) reverse engineer requirements from code, e) keep up with how the code state evolves, f) apply changes from other branches into the main branch. The surveyed developers also expressed some challenges with the usability of existing tools, such as their inability to detect file moves and renames, and their difficult configuration (e.g., setting up `git bisect` to find the commit that introduced a bug).

1.1 Motivation

Grund et al. [14] conducted a survey with 42 professional software developers and found that they prefer source code history information at the method/function and class level rather than the file level. Moreover, the tools used by the developers to inspect code history, such as `git log` and IntelliJ's history feature, are unable to find the commit that actually introduced a method and deal with complex structural changes (e.g., method moves). LaToza and Myers [26] surveyed 179 professional software developers at Microsoft and asked them to list hard-to-answer questions that they had recently asked about code. Among the collected responses, developers asked about "*Where was this variable last changed?*" when debugging, "*When, how, by whom, and why was this code changed or inserted?*" when they want to find the code's creation in history to understand its context and motivation, and finally "*How has it changed over time?*" when they want to know the entire history of a block of code, rather than its most recent change. These findings motivate the need for developing tools that can track change history at a more fined-grained level, focusing on specific program elements, such as methods/functions and variables.

Accurate code snippet tracking is also essential in many areas of software engineering research. Alencar da Costa et al. [9] pointed out that bug-inducing analysis algorithms (e.g., SZZ [23, 40, 48]) suffer from broken historical links due to file moves and renames. This further affects the results of defect prediction techniques and empirical studies investigating the characteristics of bug-introducing changes, which rely on the original SZZ algorithm or its variants [36]. Shen et al. [37] showed that automatic source code merging tools often fail to track the changed program elements correctly due to overlapping refactoring operations, and thus are unable to perform the auto-merging. The automatic migration of client software to newer library and framework versions, requires to track the updated API program elements (i.e., methods and fields) from the source to the target version, extract changes in the API signatures, and adapt accordingly the API references in client's code [8, 10, 20]. API program element tracking has been performed both at commit level [4, 5] and release level [28, 29]. However, fine-grained program element tracking at commit level may be more accurate than release level [3], as comparing directly two releases involves significantly more noise from overlapping changes performed in all commits between the two releases.

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3549079

The inherent limitations of the line-based text diff and blame tools, which are predominantly used in the aforementioned software engineering tasks, motivated researchers to develop techniques for tracking more accurately program elements, such as methods/functions and classes, in the commit history of software projects [12, 14–17, 27, 41, 42]. These techniques deal with changes that modify the name/signature or location of a program element and can cause a split in its history. Hora et al. [17] found that 25% of classes and methods have at least one *untracked* change (i.e., move, rename, extract, inline refactoring) in their histories. Despite the significant accuracy improvements brought by program element tracking tools, they still have some limitations, which we discuss in the next subsection.

1.2 Limitations of Current Tracking Tools

CodeShovel [14], is the most accurate tool for uncovering Java method histories to-date, as it produces complete and accurate commit change histories for 90% of methods, including 97% of all method changes. CodeShovel is partially *refactoring-aware*. It supports the tracking of methods with changes in their signature (e.g., method rename, parameter addition/deletion), methods whose parent file has been moved/renamed, and methods moved to another file. However, our experiments have shown that it fails to track properly methods from which a significant part of their body has been extracted to new methods, as it uses a 75% body similarity threshold to match modified methods, and thus erroneously matches the original method with the extracted one. The same limitation holds when methods with a relatively large body are inlined to the tracked methods. Our approach overcomes this limitation by being fully refactoring-aware and detecting method extractions/inlines from/to the tracked methods.

FinerGit [16] and Historage [15] create a finer-grained Git repository, in which each Java method exists in its own file, and take advantage of Git mechanisms to track changes on each individual method's corresponding file. FinerGit improves on the limited capability of Historage to track renamed or moved methods, especially for small methods, by formatting each file to include a single token from the corresponding method in each line. This formatting makes Git's line-based similarity computation mechanism more robust in matching small methods, which have been renamed or moved. Pre-processing an entire repository to place each method in its own file, is computationally expensive and requires additional hard disk space, which can be prohibitive, especially for large repositories with many files and a long commit history. As a matter of fact, Grund et al. [14] found that FinerGit ran out of memory or did not finish pre-processing within 15 minutes for the four largest repositories in their validation data set. Moreover, this pre-processing cost did not contribute an accuracy improvement, as the recall of FinerGit was 65% compared to 90% of CodeShovel [14].

Kim et al. [42] proposed an approach to identify function mappings across revisions even when a function's name changes. Their approach considers the similarity of the following factors: function name, incoming and outgoing calls, signature, function body text diff, complexity metrics and the results of two clone detection tools (CCFinder and MOSS). The computation of text diff and the execution of multiple clone detection tools may have a considerable cost, especially when there are many combinations of deleted and newly added functions to be compared.

A common limitation of all aforementioned tools is that they are designed to support only the tracking of methods, and cannot be extended to support the tracking of other program elements, such as variables and attributes, whose evolution is also interesting for the developers. Several studies have shown that developers frequently refactor variables and attributes, which makes their tracking in the commit history challenging. Negara et al. [33] found that Rename Local Variable and Rename Field are among the most popular refactorings applied by developers. Negara et al. [34] surveyed 420 developers, who ranked Change Field Type as the most relevant and applicable transformation that they perform. Ketkar et al. [22] found that developers who changed the type of a variable or attribute, they also renamed it in 55% of the examined instances.

Godfrey and Zou [12] implemented a tool, named Beagle, that can detect structural changes like rename, move, split, and merge at function, file, and subsystem level. They rely on *origin analysis* to decide if a program entity is renamed or moved and a function call analysis to discover merges and splits of program entities. Although Beagle supports the tracking of program elements at different levels of granularity (i.e., function, file, subsystem), it requires as input two complete versions of a software system in order to extract static relations between program entities (e.g., function calls), and calculate various metrics. This makes Beagle impractical for program element tracking at commit level.

Steidl et al. [41] proposed an incremental origin analysis that applies some heuristics to find moved, renamed, split, and merged source code files. In contrast to Beagle, their approach is commit-based and incrementally reconstructs the history based on clone information and file name similarity. However, the proposed origin analysis is limited to files and thus does not support the tracking of program elements, such as methods, variables and attributes.

Lee et al. [27] implemented a tool named Tempura, enabling code completion and navigation to operate on multiple revisions of code at a time. To support these features, Tempura pre-processes the commit history of a Git repository, and for each added, modified, renamed, or deleted Java file extracts and records its API information (i.e., type, method, field declarations) indexed by the enclosing type's fully qualified name. Temporal navigation is performed by a simple index lookup to list the revisions in which the selected program element changed. A major limitation of Tempura is that it requires to pre-process and index the repository under analysis, which can take several minutes, especially for large repositories. Moreover, Tempura is not fully refactoring-aware, as it infers only Class Rename and Move refactorings by leveraging Git's file rename/move detection capability.

Hora et al. [17] introduced the concept of *change graph* to model the evolution of classes, methods, and their related changes in the commit history of a project, and study the phenomenon of *untracked* changes. In this graph, each class or method is represented as a node, while each tracked or untracked change is represented as an edge between two nodes. However, Hora et al.'s change graph is limited in modelling only the evolution of classes and methods, supports a limited number of refactoring types (5 class-level and 6 method-level refactorings), and uses RefDiff [39] for the detection

of refactoring operations, which has inferior precision, recall and performance than RefactoringMiner [45, 46]. Finally, the graph edges model only a small subset of refactoring operations, while other kinds of changes, such as method body and signature changes are omitted. Thus, Hora et al.'s change graph cannot be used to find all commits where a program element changed, i.e., the graph can provide only the commits in which a program element is involved in refactorings.

Summary: None of the currently available tools can track the evolution of fields and local variables. Moreover, the evolution tracking of methods is limited, as commonly applied refactoring types, such as Extract Method, are not supported by the currently available tools. According to Negara et al. [33] and Tsantalis et al. [45], Extract Method is the most commonly applied refactoring on methods.

1.3 Contributions

Our solution offers some significant improvements over the previous state-of-the-art and novel contributions:

(1) We fix all inaccuracies that we found in the oracle provided by Grund et al. [14] including the evolution history of 200 methods. Moreover, we extend this oracle by adding the evolution history of 1345 variables declared in these methods.
(2) We support new kinds of program element changes, such as documentation and annotation changes, which are not supported by CodeShovel [14] and other tools.
(3) We improve both precision and recall in method evolution tracking over the previous state-of-the-art, CodeShovel [14].
(4) We extend RefactoringMiner [45, 46] with heuristics for performing partial and incremental commit analysis in order to reduce the execution time. We show that the applied heuristics achieve an execution time comparable to that of CodeShovel [14] without jeopardizing precision or recall.
(5) We propose the concept of *evolution hooks* as a way to model the change history of methods extracted/inlined from/to the tracked method of interest, and link the change histories of relevant program elements.
(6) We are the first to support the evolution tracking of variable declarations with 99.7% precision and 99.8% recall.

2 APPROACH

This section presents our approach for modelling and reconstructing the changes applied on program elements, such as method and variable declarations, in the commit history of a project.

2.1 Program Element Identifier

Each program element e is uniquely identified in the commit history of a software repository with the following tuple:

$$I_e = (V_e, CON_e, SIG_e) \quad (1)$$

where V_e is the version of e corresponding to the SHA-1 Git commit ID in which a change took place on e, CON_e is the signature of the container in which e belongs to, and SIG_e is the signature of e.

The typical container structure in Java programs is shown in the example of Figure 1. The container of a type declaration c is

```
Zull-core/src/main/java
  package com.netflix.zuul;
    public class FilterProcessorImpl {
      protected final FilterLoader filterLoader;
      protected ZuulFilter getErrorEndpoint(ZuulMessage msg) {
        SessionContext context = msg.getContext();
        String endpointName = context.getErrorEndpoint();
          ...
        ZuulFilter errorEndpoint = getFilterByNameAndTyper(
                          endpointName, FilterType.ENDPOINT);
        if(errorEndpoint == null) {
          String errorStr = "..." + endpointName;
          LOG.error("..."  + errorStr, context.getError());
        }
        return errorEndpoint;
      }
    }
```

Figure 1: Typical container structure in Java programs

the tuple $CON_c = (SRC_c, PKG_c)$, where SRC_c is the source folder path and PKG_c is the package name in which c belong to. It is very important to include the source folder path in the container tuple, as it is possible to have a type declaration with the same name and package in two different source folders. The container of a method declaration m is the tuple $CON_m = (CON_{C_m}, SIG_{C_m})$, where C_m is the type declaration in which m belong to, CON_{C_m} and SIG_{C_m} are the container and signature of C_m, respectively. Finally, the container of a variable/parameter declaration v is the tuple $CON_v = (CON_{M_v}, SIG_{M_v})$, where M_v is the method declaration in which v is declared, CON_{M_v} and SIG_{M_v} are the container and signature of M_v, respectively.

The signature of a type/enum/annotation declaration c is the tuple $SIG_c = (N_c, K_c, S_c, I_c, T_c, A_c, V_c, M_c)$, where N_c is the name of c, K_c is the kind of c, which is a categorical variable taking four possible values, namely *class*, *interface*, *enum* and *@interface* (for annotation type declarations), S_c is the super-class type of c, I_c is the list of super-interfaces of c, T_c is the list of type parameters of c in the case that c is a parameterized type, A_c is the list of annotations of c, V_c is the visibility of c, which is a categorical variable taking four possible values (*public*, *protected*, *private*, or *package-private*), and finally M_c is the list of modifiers of c (*final*, *static*, *abstract*). All elements in the signature of c can change during its evolution, including its kind K_c.

The signature of a method declaration m is the tuple $SIG_m = (N_m, R_m, P_m, E_m, T_m, A_m, V_m, M_m, B_m, D_m)$, where N_m is the name of m, R_m is the return type of m, P_m is the ordered parameter list of m, E_m is the list of thrown exception types of m, T_m is the list of type parameters of m in the case that m is a parameterized method, A_m is the list of annotations of m, V_m is the visibility of m, M_m is the list of modifiers of m (*final*, *static*, *abstract*, *synchronized*), B_m is the hashed value of m's body string representation, and finally D_m is the hashed value of m's Javadoc and inline comments.

The signature of a variable/parameter declaration v is the tuple $SIG_v = (N_v, T_v, A_v, M_v, S_v)$, where N_v, T_v, A_v, M_v, S_v are the name, type, annotation list, modifier (i.e., *final*) and scope of v, respectively. S_v includes all statements v is visible to, and thus v can be referenced from. The scope of a variable starts from the first statement following the declaration of the variable and ends to the last statement within the block in which the variable is declared. Figure 1 depicts the statements within the scopes of variables `endpointName` and `errorStr` in the respective rectangular boxes. The scope is essential for distinguishing variables with the same name and type declared in different blocks of a method.

2.2 Tracking Process

Our solution relies on RefactoringMiner [45] to track a program element in the commit history of a project, and report all changes and refactoring operations performed on it. Despite the fast execution time of RefactoringMiner (44 ms on median and 253 ms on average per commit), running it on the entire commit history of the project is computationally inefficient, as the tracked program element is changing in a relatively small subset of commits, and furthermore it is not always necessary to analyze all modified files in a commit to track a single program element, especially in large commits involving thousands of modified files. Therefore, we developed some heuristics and extended RefactoringMiner to perform partial and incremental commit analysis.

Input: Similarly to CodeShovel, our tool is using as input a Git repository URL, a starting commit SHA-1 ID (or HEAD by default), the file path containing the program element of interest, the name of the program element, and the start line number of the program element's declaration. Both name and start line are needed to disambiguate the program element of interest, as it is possible to have multiple program elements with the same name (i.e., overloaded methods, identically named variables declared in different blocks of the same method), and it is possible to declare two or more variables/parameters on the same line.

Output: A graph in which the nodes represent program elements with their unique identifiers (i.e., program elements in different commits), and each edge connecting two nodes includes the list of changes between the corresponding program elements. The edges are directed from *child* commit program elements to the matching *parent* commit program elements. In other words, by traversing the output graph, we visit previous versions of a program element (i.e., *backward* tracking). We decided to model the output as a graph, because it is possible to have forks. For example, when multiple methods from different subclasses are pulled up in a single superclass method, the same program element (i.e., superclass method) is connected with multiple program elements (i.e., subclass methods). In addition, we model extracted and inlined methods as branches in the evolution graph of the main tracked program element, as we will explain in Section 2.3. To facilitate the comparison with CodeShovel [14], we can transform the graph output to a list of commits in which the input program element changed along with the corresponding kinds of changes in each commit.

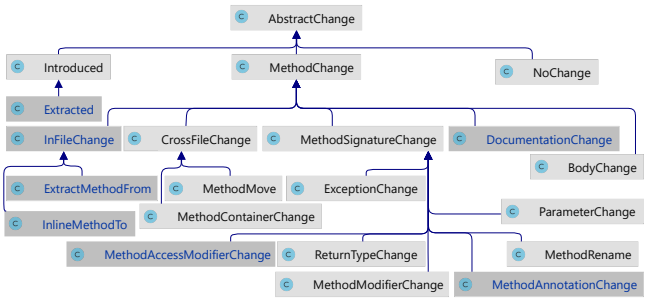

Figure 2: Hierarchy of supported change kinds for methods adopted by CodeShovel [14]. Newly supported change kinds are highlighted in blue colour.

We adopted and extended the change hierarchy supported by CodeShovel [14] for method tracking, as shown in Figure 2. The newly supported change kinds are highlighted in blue colour, and deal with *InFileChange*, i.e., the extraction of a new method from the body of the tracked method, and the inline of a method within the body of the tracked method. In addition, we support two more kinds of changes, namely *AnnotationChange*, and *DocumentationChange*. The latter involves changes in the *Javadoc* or *inline* comments within the body of the tracked method. Finally, for some change kinds we have a more fine-grained reporting. For example, CodeShovel reports any change(s) in the parameter list of a method as a single *ParameterChange*, while we report individually for each parameter the following fine-grained changes, *Add*, *Remove*, *Rename*, *ChangeType*, *Merge*, *Split*, and *Reorder*.

Figure 3 shows the change hierarchy supported by our tool for variable tracking. *InFileChange* supports the scenario of a variable declaration being moved to another method in the same container, as part of a code fragment extracted to a new method or inlined from a previously existing method, while *CrossFileChange* supports the scenario of a variable declaration belonging to a method moved to another container.

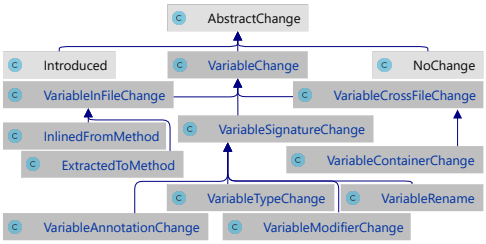

Figure 3: Hierarchy of supported change kinds for variables.

An overview of the tracking process is shown in Figure 4 and consists of the following steps:

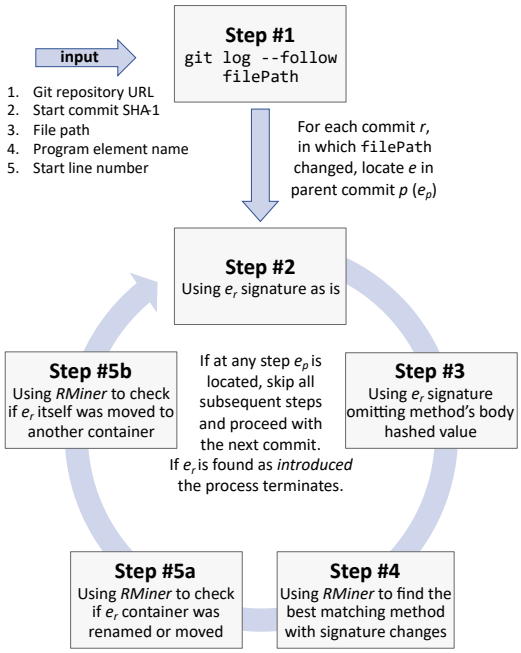

Figure 4: Overview of the tracking process steps.

Step #1: Given the input file path in which program element e is located, we first find all commits in the project's history in which file path is modified using the command `git log --follow filePath`. The `--follow` option is particularly important, as it continues listing the history of a file even when it gets renamed or moved. The first step is based on the assumption that if there are no changes in the file containing e in a given commit, then we can skip the analysis of this commit as e has no changes too.

Step #2: For each commit r in the subset of commits obtained from Step #1, we create a partial source code model for r and p (i.e., r's parent commit) by parsing only the source file corresponding to the input file path. If r is the starting commit, then we locate program element e in r's model using its name and start line, and construct its signature (sig_{e_r}) and container (con_{e_r}) as explained in Section 2.1. If r is a subsequent commit, then we have e_r's signature and container from the previous iteration of the tracking process (i.e., the matched program element from the previously processed commit). Then, we attempt to locate a program element with the same signature and container in p's model. If a match is found, then we link the two program elements (e_r, e_p) with their unique identifiers and report *NoChange*. Such a match is possible when all containers of the tracked program element up to the root have identical signatures, but the file has changes in other irrelevant parts. In such case, there is no need to execute RefactoringMiner. If no match is found, we relax the comparison of signatures, as explained in the next step.

Step #3: If program element e itself or its container is a method (i.e., e is a variable), then we omit B_m (i.e., the hashed value of the method's body string representation) from the method's signature tuple, and we attempt to locate a program element with the same relaxed signature and container in p's model.

If a match is found and e is a method, we link the two methods (e_r, e_p) with their unique identifiers and report *BodyChange*. Such a match is possible when there are changes in the body of the tracked method, but its signature remains unchanged. However, we still need to check if the tracked method is involved in an Extract Method or Inline Method refactoring. To avoid an unnecessary execution of RefactoringMiner, we extract all method calls from e_r and e_p, respectively, and keep the calls that do not have a caller expression or have `this` as a caller expression (i.e., the calls to local methods). If there are additional calls in e_r's call list, then we need to check if a local method was extracted from e_p. If there are additional calls in e_p's call list, then we need to check if a local method was inlined to e_r. In both cases, we execute RefactoringMiner on the partial source code models for commits r and p including only T_r (i.e., the type declaration containing e_r) and T_p (i.e., the type declaration containing e_p). For each Extract Method or Inline Method refactoring returned by RefactoringMiner, we introduce the extracted/inlined program elements as *evolution hooks* in the output graph (Section 2.3 includes more details). However, if the two method call lists are identical, then there is no need to execute RefactoringMiner.

If a match is found and e is a variable, we link the two variables (e_r, e_p) with their unique identifiers and report *NoChange*. Such a match is possible when e_r and e_p still have the same name, type, modifier, annotations, and statements in their scopes, despite the changes in the body of their container method. However, if no matching variable is found, but the container methods con_{e_r} and con_{e_p} are matched, we extract all variables declared within the body of con_{e_r} and con_{e_p} and omit S_v (i.e., the list of statements within the variable's scope) from all variable signature tuples. If the two lists of relaxed variable signatures are identical, then we link e_r with the corresponding variable e_p (i.e., the variable having the same position in the list of variables declared within con_{e_p}). Such a match is possible when e_r and e_p still have the same name, type, modifier, annotations, but there are some syntactic changes in the statements within the scope of the variables. However, if the two lists of relaxed variable signatures are not identical, then we execute RefactoringMiner on the partial source code models for commits r and p including only T_r (i.e., the type declaration containing con_{e_r}) and T_p (i.e., the type declaration containing con_{e_p}). RefactoringMiner will initially match the container methods con_{e_r} and con_{e_p} and perform a thorough analysis after matching the statements within their bodies, in order to find *renamed*, *inlined*, *extracted*, *split*, *merged*, *moved* (due to method extraction or inline), *added*, *deleted*, and *matched* variables. We locate e_r in the reported refactorings, link e_r with the corresponding variable e_p, and report all changes found between them.

If by the end of Step #3 no match is found for e, this is an indication that there are major changes in the signature of e itself or its container. This scenario is addressed in the next step.

Step #4: Assuming that e_r is contained within type declaration T_r in commit r, and exists a type declaration T_p in commit p, where both T_r and T_p have an identical name and container signature, then we attempt to locate e_p within T_p, by executing RefactoringMiner on the partial source code models including only T_r and T_p. RefactoringMiner initially matches the method pairs with identical signatures (i.e., method name and parameter types), and then compares all combinations of the remaining unmatched methods from T_r with the remaining unmatched methods from T_p to find the best matching method pairs with changes in their signatures [45].

If e_r is a method, we check if there exists a pair (e_r, e_p) in the best matching method pairs. If so, we link the two methods (e_r, e_p) with their unique identifiers and report all changes in their signatures and bodies. In addition, we add any local methods extracted from e_p or inlined to e_r as *evolution hooks* in the output graph (Section 2.3), as this information is provided by RefactoringMiner when comparing two type declarations.

If e_r is a variable, we check if there exists a pair (con_{e_r}, con_{e_p}) including the container of e_r in the best matching method pairs. If so, we retrieve all variable-related refactorings (i.e., *rename*, *inline*, *extract*, *split*, *merge*, *move*, *add*, *delete*, and *match*) extracted by RefactoringMiner for this pair of methods. We locate e_r in the reported refactorings, link e_r with the corresponding variable e_p, and report all changes found between them.

If by the end of Step #4 there is still no match found for program element e, this is an indication that either e itself or its container has been moved to another file, or the type declaration T_r containing e_r has been renamed or moved to another package. This scenario is addressed in the next and final step.

Step #5: At this stage, we keep the partial source code model including only T_r for commit r, but add all modified and removed files

in commit p to p's source code model (i.e., we create the complete source code model for commit p). Then, we execute RefactoringMiner on these two source code models, and collect all reported refactorings, including Rename Class, Move Class and Move Method. Step #5 is the most time-consuming step in the tracking process, as we include all modified files in commit p. During our experiments, we discovered some commits (e.g., in project hadoop [30]) in which the developers moved thousands of source code files from one source folder to another (i.e., the moved files have a change in their file path, but their contents remain identical), or simply re-organized the import declarations (e.g., hibernate-orm [47]), or updated comments, such as the license header (e.g., eclipse-jetty [11]). To avoid the unnecessary processing of files and speed-up the tracking process, we exclude from p's source code model all files with identical contents, and files with only trivial changes in comments and import declarations [21]. Finally, we support three scenarios in which additional files need to be included in r's source code model to correctly track program element e.

(1) Changes on e can only be inferred from changes in other program elements: RefactoringMiner infers signature-level refactorings for method pairs not having a body (i.e., *interface* or *abstract* methods), or method pairs that could not be matched based on statement mapping information (i.e., methods with large differences in their bodies due to functionality changes) from the refactorings/changes detected on method pairs having identical signatures with the unmatched method pairs [45]. The intuition is that a change in the signature of an *abstract* or *interface* method should propagate to all concrete implementations of that method (i.e., overriding methods). Assuming that e_r is contained within type declaration T_r in commit r, we get the extended superclass and implemented interface types of T_r and check for each one of these types if it corresponds to a modified file in commit r to ensure the super type is a local type declaration of the analyzed system. If a super type S_r indeed corresponds to a modified file in commit r, then we use regular expressions to check if other modified files in commit r extend or implement S_r and add them to r's source code model. This approach enables the inference mechanism of RefactoringMiner with the least possible computation cost. For example, in project OkHttp [49], the method pair `synStream—headers` in inner class `SpdyConnection.Reader` is matched by additionally including class `MockSpdyPeer.InFrame` to r's source code model, as both classes implement the `FrameReader.Handler` interface.

(2) e is copied into a new file: In some projects, which are libraries with public APIs, we found that developers tend to copy the methods they want to deprecate into a new file, and then declare the original methods or their container class as `@deprecated`. Let us assume that e_r is copied in type declaration T_r in commit r from type declaration T'_p in commit p. Without additionally including the original type declaration containing the copied method T'_r to r's source code model, then e_r would be detected as *moved* from T'_p to T_r, instead of *introduced* in T_r as a new method. To address this issue we use a regular expression to check if other modified files in commit r include a `@deprecated` annotation with a `@link` to e_r's signature (e.g., `copy` methods copied from `IOUtils` to `CopyUtils` in project commons-io [31]), or a `@deprecated` annotation with a reference to T_r name (e.g., deprecated classes `IOUtil` and `EndianUtil` referring to newly added classes `IOUtils` and `EndianUtils`, respectively, in project commons-io [32]) and add them to r's source code model. Moreover, we check if other modified files in commit r have the same name as T_r, but different package (e.g., methods copied from deprecated class `org.apache.commons.lang.NumberUtils` to new class `org.apache.commons.lang.math.NumberUtils` in project commons-lang [7]) and add them to r's source code model.

(3) e is extracted to a new file: In this scenario, developers move some members of an existing class into a new class, and instantiate the new class into the origin class in order to access the moved functionality (i.e., Extract Class refactoring), or extend the origin class in order to inherit the non-moved functionality (i.e., Extract Subclass refactoring). Let us assume that e_r is moved in type declaration T_r in commit r from type declaration T'_p in commit p. Without additionally including the original type declaration containing the moved method T'_r to r's source code model, then T'_p would be detected as *renamed* to T_r (if multiple members from T'_p have been moved to T_r), instead of T_r being extracted from T'_p, and T'_r being matched with T'_p. To address this issue we use a regular expression to check if other modified files in commit r create an instance of T_r (e.g., methods moved to extracted class `SourceFileInfoExtractor` from class `ProjectResolver` in project javaparser [43]), or are extended by T_r (e.g., methods pushed down to extracted subclass `AbstractNestablePropertyAccessor` from origin class `AbstractPropertyAccessor` in project spring-framework [35]) and add them to r's source code model.

Step #5a: Assuming that e_r is contained within type declaration T_r in commit r, we check all class-related refactorings (i.e., *rename*, *move* class) to find a pair of type declarations (T_r, T_p) involving T_r. If such a pair is found, we obtain the corresponding class-level diff object from RefactoringMiner, which includes all pairs of matched methods.

If e_r is a method, then we check if there exists a pair (e_r, e_p) in the matching method pairs. If so, we link the two methods (e_r, e_p) with their unique identifiers, and report a *FileMove* change (i.e., T_r is renamed/moved to T_p) in addition to any changes in their signatures and bodies. Moreover, we add any local methods extracted from e_p or inlined to e_r as *evolution hooks* in the output graph (Section 2.3), as this information is provided by RefactoringMiner when comparing two type declarations.

If e_r is a variable, we check if there exists a pair (con_{e_r}, con_{e_p}) including the container of e_r in the matching method pairs. If so, we retrieve all variable-related refactorings (i.e., *rename*, *inline*, *extract*, *split*, *merge*, *move*, *add*, *delete*, and *match*) extracted by RefactoringMiner for this pair of methods. We locate e_r in the reported refactorings, link e_r with the corresponding variable e_p, and report a *MovedWithMethod* change (i.e., con_{e_r} has been relocated to con_{e_p}) in addition to any changes found between the variables.

If by the end of Step #5a there is still no match found for program element e, this is an indication that either e itself or its container has been moved to another file.

Step #5b: Assuming that e_r is contained within type declaration T_r in commit r, we check all method-related refactorings involving moves (i.e., *move*, *pull up*, *push down* method) to find a pair of method declarations (e_r, e_p) if e_r is a method, or (con_{e_r}, con_{e_p}) if

e_r is a variable, involving a move from T_r. If such a pair is found, we proceed in the same way as described in Step #5a with the only difference being the report of a *MethodMove* change instead of a *FileMove* (if e_r is a method).

If by the end of Step #5b there is still no match found for program element e, we report that e_r has been *Introduced* in commit r. We further examine the refactorings reported by RefactoringMiner to find if e_r is introduced by an Extract Method refactoring, and add the method(s) from which e_r is extracted as *evolution hooks* in the output graph (Section 2.3). If a matching program element e_p is found throughout Steps #2 to #5, we use its signature (sig_{e_p}) and container (con_{e_p}) to continue tracking program element e in the remaining commits obtained from Step #1.

2.3 Change Graph Evolution Hooks

It is very likely that a developer would like to inspect the evolution of program elements which are extracted from or inlined to the main tracked program element. The intuition behind this feature is that the extracted/inlined program elements were originally part or became part of the tracked program element at some point in its evolution. To support this feature our program element tracking process (Section 2.2) introduces the extracted/inlined program elements as *evolution hooks* in the change graph of the main tracked program element. The developer can attach on demand the evolution sub-graph of an extracted/inlined program element by expanding the corresponding *evolution hook*.

Figure 5 shows how we model the evolution of an extracted program element on the change graph of a tracked program element. Assuming an Extract Method refactoring takes place in commit r, then we create a node for the extracted element e'_r with a unique identifier $I_{e'_r}$ including the commit ID of r, since e'_r starts existing in commit r, its signature ($sig_{e'_r}$) and container ($con_{e'_r}$) constructed as explained in Section 2.1. When the developer decides to expand the e'_r node, we use $I_{e'_r}$ and start commit r as input for our tracking process, which is executed *forwards* in this case (i.e., from parent commit to child commit), and attach the resulting graph on the e'_r node, as shown in Figure 5.

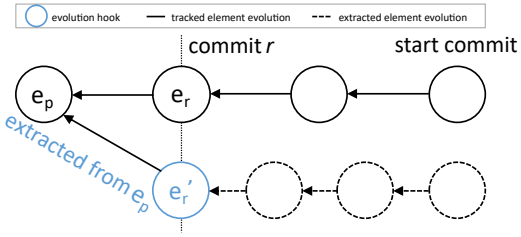

Figure 5: Tracking of a program element extracted from the tracked element.

Figure 6 shows how we model the evolution of an inlined program element on the change graph of a tracked program element. Assuming an Inline Method refactoring takes place in commit r, then we create a node for the inlined element e'_p with a unique identifier $I_{e'_p}$ including the commit ID of p (i.e., the parent commit of r), since e'_p last exists in commit p, its signature ($sig_{e'_p}$) and container ($con_{e'_p}$) constructed as explained in Section 2.1. When the developer decides to expand the e'_p node, we use $I_{e'_p}$ and start commit p as input for our tracking process, and attach the resulting graph on the e'_p node, as shown in Figure 6.

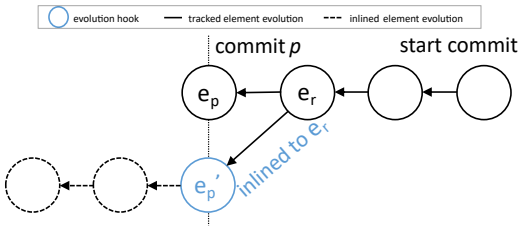

Figure 6: Tracking of a program element inlined to the tracked element.

It is also very likely that a developer would like to inspect the evolution of the method(s) from which the tracked program element is extracted. Figure 7 shows how we model the evolution of a program element, which is the origin of extraction for a tracked program element. Assuming an Extract Method refactoring takes place in commit r, then we create a node for the origin element e'_p with a unique identifier $I_{e'_p}$ including the commit ID of p (i.e., the parent commit of r), since e_r was originally contained in e'_p in commit p, its signature ($sig_{e'_p}$) and container ($con_{e'_p}$) constructed as explained in Section 2.1. When the developer decides to expand the e'_p node, we use $I_{e'_p}$ and start commit p as input for our tracking process, and attach the resulting graph on the e'_p node, as shown in Figure 7.

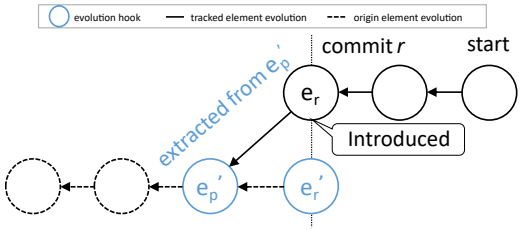

Figure 7: Tracking of a program element from which the tracked element is extracted.

3 EVALUATION

In our evaluation, we investigate the following research questions:

RQ1. What is the accuracy of CodeTracker in method tracking and how does it compare to that of CodeShovel?

RQ2. What is the accuracy of CodeTracker in variable tracking?

RQ3. How does the execution time of CodeTracker compare to that of CodeShovel?

RQ4. What is the execution time speedup of CodeTracker over the default operation mode of RefactoringMiner?

3.1 Oracle Update and Extension

Grund et al. [14] created an oracle with the change history of 200 methods from 20 popular open-source project repositories. In particular, they used 100 of these methods (*training set*) to optimize the threshold values used in CodeShovel, until they achieved 100% training accuracy, and the remaining 100 methods (*testing set*) to

validate the accuracy of CodeShovel. We decided to use both the training and testing sets to evaluate the accuracy of our tool and compare with that of CodeShovel, since Grund et al. spent 100 hours of manual validations to construct their oracle, and thus we can consider it as very reliable. However, after executing our tool we found major differences in the commit history of some methods. After careful inspection, we found out that 18 methods from the *training set* and 9 methods from the *testing set* were matched with a method extracted from their body at some point in their change history. As a result, after the commit in which the the originally tracked method is erroneously matched with an extracted one, the oracle includes the change history of the extracted method, instead of the original method. In all these cases, a significant portion of the originally tracked method (over 75%, or even the entire method body) is extracted to a new method. Silva et al. [38] found that developers in many cases extract a large portion or even the entire body of a method into a new one, either to introduce an alternative method signature (i.e., different input/output parameter types), or to deprecate a method that is no longer needed. The original method remains in the code base and delegates to the extracted one in order to preserve the public API (i.e., backwards compatibility). Despite the strong similarity of the extracted method with the original method, it is not correct to match them, as the original method still remains in the code base with an identical signature (i.e., method name and parameter types) in most cases, and thus should be further tracked. As we discussed in Section 2.3, we model such cases as branches in the change graph of the tracked method using *evolution hooks*, and continue tracking the changes on the original method. Moreover, we found some cases where the oracle reports a method being moved to another file (i.e., *MethodMove* change), while in reality the type declaration that contains the method has been renamed or moved, and thus a *FileMove* change is the correct one.

We corrected all discrepancies found in the oracle by removing the false change instances due to incorrect method matches and adding the new change instances resulting after the correction of method matches. All removed and new change instances have been manually validated by inspecting the changes on GitHub. Table 1 provides a detailed overview of the changes being *common* with the original oracle (C columns), the changes *removed* from the original oracle (R columns), and the *new* changes added to the original oracle (N columns) for both training and testing sets.

We further extended this oracle with the change history of the local variables and parameters declared in these 200 methods. Since their number is large (967 variables in the *training set* and 378 variables in the *testing set*), we decided to follow a semi-automated approach to construct the oracle, instead of sampling a small number of variables and manually tracking their changes in the commit history. We leverage information from the method tracking oracle, as we know for sure that the commits in which a variable changed is a subset of the commits in which its container method changed. Next, for each variable, we execute our tool to perform backward tracking on the commits in which the container method of the variable changed. The output is a subset of these commits along with the changes found in each commit for the tracked variable. There are two possible termination conditions:

Table 1: Updates in the oracle created by Grund et al. [14]
C: common R: removed N: new

	Training set			Testing set		
Change Kind	**C**	**R**	**N**	**C**	**R**	**N**
Body Change	2276	234	29	459	68	26
File Move	238	24	27	160	7	27
Parameter Change	220	30	6	68	19	8
Return Type Change	47	10	4	15	2	1
Modifier Change	44	12	2	16	4	1
Exception Change	40	9	0	5	3	2
Rename	14	8	7	16	6	2
Method Move	19	23	3	14	27	3
Introduced	81	18	19	83	17	17
Documentation Change	—	—	439	—	—	93
Annotation Change	—	—	42	—	—	24
Total	2979	368	578	836	153	204

(1) The variable tracking reaches the commit in which the container method was introduced. This means that the variable exists since the introduction of its container method.
(2) The variable is introduced in a commit before reaching the container method introduction commit. This means that the variable was added as part of new functionality implemented in the container method, or because some other method was inlined to the container method.

We validate all reported changes by manually inspecting the corresponding commits. If a reported change is correct, we add it in the oracle. If a reported change is incorrect, this means that there was a wrong variable match performed by our tool. In that case, we manually determine the correct match/change for the tracked variable, add it in the oracle, and continue the tracking process from the parent commit for the correctly matched variable. This process is repeated until we reach one of the two termination conditions for each tracked variable. Reaching a termination condition guarantees that we have no missing variable changes in our oracle. The number of instances per change kind for the variables are shown in Table 2.

Table 2: Number of instances per change kind for variables.

Change Kind	Training set	Testing set
Introduced	967	378
Rename	145	27
Annotation Change	13	4
Type Change	264	67
Modifier Change	123	52
Total	1512	528

3.2 RQ1: Method Tracking Accuracy

The precision and recall of CodeTracker and CodeShovel was computed at two levels of granularity, namely *commit level* (i.e., finding the commits in which a method changed), and *change level* (i.e., finding the kinds of changes that occurred in the commits in which a method changed). When there are only changes in inline comments within the body of a method, CodeShovel reports a *BodyChange*.

To ensure a fair comparison, we considered such cases as true positives, despite being labelled as *DocumentationChange* in the updated oracle. Moreover, for any kind of change in the parameter list of a method, CodeShovel reports a *ParameterChange*, while CodeTracker reports more fine-grained parameter changes, such as *Add*, *Remove*, *Rename*, *ChangeType*, *Merge*, *Split*, *Reorder*, *Add/Remove Modifier*, and *Add/Remove/Modify Annotation*. We also considered the coarse-grained *ParameterChange* reports as true positives.

Table 3: Method tracking precision/recall at commit level

Dataset	Tool	TP	FP	FN	Precision	Recall
Training	CodeShovel	2751	219	74	92.63	97.38
	CodeTracker	2825	1	0	99.96	100
Testing	CodeShovel	776	68	65	91.94	92.27
	CodeTracker	839	0	0	100	100
Overall	CodeShovel	3527	287	139	92.48	96.21
	CodeTracker	3664	1	0	99.97	100

Based on the results shown in Table 3, our tool, CodeTracker, has a consistent performance in both *training* and *testing* sets at commit level, with an overall precision of 99.97% and recall of 100%. On the other hand, CodeShovel has a lower precision and recall on the *testing set* compared to the *training set*, as the similarity thresholds used internally by CodeShovel were optimized on the *training set*. This is an inherent limitation of approaches relying on code similarity thresholds, as it is very difficult to derive *universal* threshold values that can work well for all projects, regardless of their architectural style, application domain, and development practices [45].

Table 4: Method tracking precision/recall at change level

Dataset	Tool	TP	FP	FN	Precision	Recall
Training	CodeShovel	3412	304	145	91.82	95.92
	CodeTracker	3557	3	0	99.92	100
Testing	CodeShovel	915	136	125	87.06	87.98
	CodeTracker	1037	3	3	99.71	99.71
Overall	CodeShovel	4327	440	270	90.77	94.13
	CodeTracker	4594	6	3	99.87	99.93

Based on the results shown in Table 4, our tool, CodeTracker, has a similar performance at change level as commit level, with an overall precision of 99.87% and recall of 99.93%. On the other hand, at change level, CodeShovel has a decrease of around 2% in both precision and recall compared to the commit level results. The FPs and FNs of CodeShovel are mainly due to mismatching 18 methods from the *training set* and 9 methods from the *testing set* with a method extracted from their body at some point in their change history. As a result, the majority of the FPs are body and signature changes found in the history of these 27 mismatched extracted methods, while the majority of the FNs are body and signature changes missed from the remaining history of the 27 original methods, in addition to missed changes in method annotations and Javadoc, which are not supported by CodeShovel. The 3 FPs for CodeTracker in the *training set* are due to a merge method scenario in project javaparser [43], for which RefactoringMiner reports methods `solve(Node)` and `solveField(Node)` to be moved from `ProjectResolver` to the same method `solve(Node)` in class `SourceFileInfoExtractor`. Although the reported moves are technically correct, we considered one of them as the correct one, since our method evolution model supports only *one-to-one* method mappings, similar to CodeShovel [14]. The remaining FPs and FNs for CodeTracker are *MoveMethod* changes misreported as *FileMove* by RefactoringMiner, because the child commit model did not include the origin file of the method (i.e., the three heuristics applied in Step #5 did not match the origin file of the method).

RQ1 finding: CodeTracker exhibits 99.9% precision and recall at both commit and change levels. Compared to the previous state-of-the-art tool, CodeTracker achieves an increase of +7.5% in precision and +3.8% in recall at commit level, and an increase of +9.1% in precision and +5.8% in recall at change level.

3.3 RQ2: Variable Tracking Accuracy

Table 5 shows the precision and recall of CodeTracker at *commit* and *change* level. RefactoringMiner detects variable-related refactorings based on the AST node replacements found in the pairs of mapped variable declaration statements/expressions between two code fragments (i.e., the body of a method in the child and parent commits). It further verifies the matched variable declaration pairs by examining the presence of statement mappings referencing the paired variables within the variable scopes. As a result, the FPs are due to incorrect statement mappings, while the FNs are due to its inability to match some statement pairs.

Table 5: Variable tracking precision/recall for CodeTracker

Dataset	Level	TP	FP	FN	Precision	Recall
Training	Commit	1459	2	1	99.86	99.93
	Change	1510	4	2	99.74	99.87
Testing	Commit	512	1	0	99.81	100
	Change	527	3	1	99.43	99.81
Overall	Commit	1971	3	1	99.85	99.95
	Change	2037	7	3	99.66	99.85

For example, in project JavaParser [44], variable `parent` is erroneously matched with variable `variableDeclarator` in the parent commit (false positive), because the initializers of the two variable declarations are identical, despite having a different type and name. However, variable `parent` is extracted as a temporary variable from `variableDeclarator`'s initializer in the child commit, in order to throw an exception if the `parent` reference is not an instance of the `VariableDeclarator` type. Therefore, variable `parent` is a newly introduced variable, and the pair of `variableDeclarator` variables is not matched (false negative). Although RefactoringMiner is able to detect Extract Variable refactorings, this particular instance was missed. In project Hadoop [13], variable `resourceSecondsMap` is erroneously matched with variable `memorySeconds` in the parent commit (false positive), instead of being detected as a newly introduced variable (false negative). However, the developer merged

two `long` variables, namely `memorySeconds` and `vcoreSeconds` into the `resourceSecondsMap`, which is a map data structure with `Long` values.

RQ2 finding: CodeTracker exhibits 99.8% precision and 99.9% recall at commit level, and 99.7% precision and 99.8% recall at change level in tracking the change history of variables.

3.4 RQ3: Execution Time

Figure 8 shows the execution time of CodeTracker and CodeShovel for tracking the entire change history of each method in the training and testing sets, respectively (the y-axis is in logarithmic scale and the units are in milliseconds). Each tool was executed separately on the same machine with the following specifications: Intel Core i7-8565U CPU @ 1.80GHz × 8, 16 GB DDR3 memory, 512 GB SSD, Ubuntu 20.04.2 LTS operating system, and Java 11.0.11 x64 with a maximum of 8GB Java heap memory (i.e., `-Xmx8g`). All 20 project repositories from the training and testing sets were locally cloned before running the tools. For each tool, we recorded the total time taken for tracking a method in its entire commit change history, including the time taken for parsing the source code files and detecting the changes on the tracked method in each commit, using the `System.nanoTime` Java method. On median, CodeShovel processes the commit change history of a method in 2 seconds, while our tool, CodeTracker, takes 3.35 seconds (1.675 times slower). On average, CodeShovel processes the commit change history of a method in 3.4 seconds, while CodeTracker takes 5.5 seconds (1.62 times slower). However, CodeTracker has a more consistent performance between the *training* and *testing* sets with similar median and average execution times. This consistent performance can be explained from the observation that CodeTracker has a slightly higher percentage of commits reaching Step #5 (i.e., the most time consuming step of the tracking process) in the *testing set* compared to the *training set* (6.13% vs. 4.24% as shown in Table 6), but at the same time has a much higher percentage of commits whose processing completes in Step #2 (i.e., the least time consuming step of the tracking process) in the *testing set* compared to the *training set* (82.33% vs. 67.52% as shown in Table 6). In other words, the *testing set* includes more commits in which the tracked method is moved or its container type declaration is moved/renamed, but at the same time has much more commits in which the tracked method remains unchanged.

Table 6: Average percentage of commits processed in each step of the tracking process.

Dataset	Step #2	Step #3	Step #4	Step #5
Training	67.52%	24.21%	3.99%	4.24%
Testing	82.33%	8.66%	2.82%	6.13%

On the other hand, CodeShovel has a longer average execution time on the *testing set* compared to the *training set*, despite the fact that the *testing set* has shorter commit change histories (34.5 and 47.5 commits on median and average, respectively) than the *training set* (73 and 86.3 commits on median and average, respectively). More specifically, CodeShovel has a larger than 10 seconds execution time for 10 methods of the *testing set* (8 of which are from project `intellij-community`), and only one method of the *training set*. Typically, the methods with longer execution times have multiple commits in which their container type declaration is moved/renamed (CodeShovel's phase 3), or the tracked method is moved (CodeShovel's phase 4). Both phases 3 and 4 are the most time consuming for CodeShovel, as it widens its search to consider all other files that were modified in a commit.

RQ3 finding: Despite CodeTracker having a slower execution time than CodeShovel (1.6 times on median and average), both tools have execution times within the same order of magnitude. The median and average execution time of CodeTracker is 3.35 and 5.5 seconds, respectively, which is acceptable given the considerably increased precision and recall over CodeShovel, the additional computation of *evolution hooks* (Section 2.3), and the parallel tracking of the local variables and parameters declared in the tracked method (i.e., CodeTracker can perform parallel tracking of a method and its variable declarations without an additional computational cost). The achieved execution time can warrant applications in both research (e.g., large-scale MSR and software evolution studies) and practice (e.g., *blame*-like tracking of method and variable change history within the context of maintenance and program comprehension tasks).

3.5 RQ4: Speedup over RefactoringMiner

To evaluate the speedup achieved through steps 2–5 of our approach, we executed RefactoringMiner with its default operation mode (i.e., with two complete source code models as input, including all modified/added files in commit r and all modified/removed files in commit p) right after step #1. The median execution time of RefactoringMiner for tracking the entire change history of a method was 8.76 and 12.29 seconds, while the average execution time was 32.83 and 42.41 seconds on the *training* and *testing* sets, respectively. Therefore, the speedup of our approach is between 2.6–3.4 times on the median execution time, and between 5.9–7.8 times on the average execution time. This is achieved without jeopardizing accuracy, as almost the same precision and recall is achieved in both ways (with the exception of very few false positives/negatives discussed in Section 3.2).

RQ4 finding: CodeTracker achieves a speedup of 2.6–3.4 times on the median, and between 5.9–7.8 times on the average execution time over the default operation mode of RefactoringMiner with a negligible effect on precision and recall. This shows that the heuristics applied throughout steps 2–5 are very effective.

4 LIMITATIONS AND THREATS TO VALIDITY

Language specificity: CodeTracker depends on RefactoringMiner 2.0 [45] for the detection of refactorings and changes on the tracked program element, which limits its applicability to Java programs. However, recently there have been efforts to extend RefactoringMiner for supporting other programming languages, e.g., Python [1, 2] and Kotlin [24, 25]. With respect to CodeTracker, supporting another language would simply require to adjust the program element signature definitions and the regular expressions used in Step #5 to the characteristics and structure of this particular language.

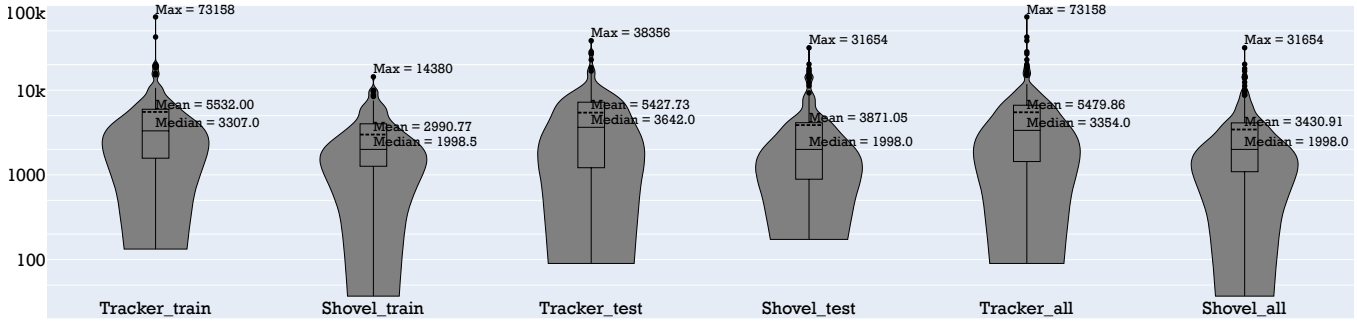

Figure 8: Execution time in milliseconds of CodeTracker and CodeShovel on training and testing sets.

Internal validity: The main threat to the internal validity is related to the construction of the oracle used for evaluating the precision and recall. To mitigate this threat we relied on an existing oracle, which was constructed by Grund et al. [14] after manually validating the change history for 200 methods (100 hours of validation). We further corrected all discrepancies found in the oracle caused by 27 methods (18 from the *training* and 9 from the *testing* set) being mismatched with a method extracted from their body, by manually inspecting and validating all new change instances that resulted after correcting the method matches. Moreover, we validated and added two new kinds of method changes (i.e., annotation and documentation changes) that were not previously supported. Based on the updated (and highly reliable) method history oracle, we constructed the change history of the variables declared in the body of these 200 methods following a semi-automated approach, as explained in Section 3.1, and manually inspecting all change instances. Overall, the manual validation effort for correcting and extending the Grund et al. oracle was approximately two person-months.

External validity: Our experiments were conducted on a relatively small dataset including the change history of 200 methods from 20 different open source projects (i.e., 10 methods from each project), which might affected the generalizability of our findings. However, we decided to rely on the same dataset used for the evaluation of CodeShovel [14] to ensure a fair comparison between the two tools. Moreover, constructing an oracle from scratch with methods from other projects would involve a lot of manual effort and probably include more errors from incorrect validations. On the other hand, our extended oracle was essentially validated by two independent research groups, which makes it more reliable.

Regarding change kinds, we considered all possible changes that can be performed on method and variable declarations with the exception of changes in the initializer of variables, as the number of changes for this particular kind was very large, and manually inspecting all of them was prohibitive.

5 CONCLUSIONS AND FUTURE WORK

In summary, the main conclusions and lessons learned are:

(1) CodeTracker has high accuracy in tracking both methods (99.9% precision/recall) and variables (99.7% precision, 99.8% recall).
(2) The proposed heuristics for setting up RefactoringMiner to perform partial commit analysis, resulted in an execution time comparable to that of CodeShovel (i.e., CodeTracker is slower by 1.67 times on median and 1.62 times on average).
(3) The speedup over the default operation mode of RefactoringMiner is 2.6–3.4 times on the median, and between 5.9–7.8 times on the average execution time. Despite this considerable speedup, almost the same precision and recall is achieved in both ways with the exception of very few false positives/negatives.

A natural extension for CodeTracker is to support attribute tracking, since RefactoringMiner detects attribute-related refactorings in a similar way as variable-related refactorings. Furthermore, given the remarkable performance of RefactoringMiner in variable tracking (i.e., mapping variable declaration statements), we are confident that a similar approach for tracking code blocks (e.g., loops, `try-catch` blocks) would be also effective. RefactoringMiner applies a bottom-up statement mapping process, where it first matches the leaf statements (i.e., statements without a body) and then uses this information to match the parent composite statements (i.e., statements with a body). This approach leads to more accurate composite statement mappings. Moreover, RefactoringMiner supports the mapping of traditional `for` loops and `if` conditionals to Java Stream API calls (e.g., `forEach()`, `map()`, `filter()`, `collect()`), and traditional `try-catch` blocks with the newer `try-with-resources` statement. This would allow to track the commit change history of loops and `try-catch` blocks, even if they have evolved into newer language constructs. Since developers are interested in the change history of methods [14], it would make sense to provide more fine-grained historical information about specific code blocks with changes in their body and conditional expressions.

Finally, we plan to conduct a usability study with professional developers by providing CodeTracker's functionality as an IDE plugin (the tool is currently available as a Java library). It is particularly interesting to investigate under which maintenance scenarios developers find the tool useful, and what kind of historical information (e.g., the kind of structural change, the author and time of the change, the commit message and linked issues) they find more relevant to their needs in each scenario.

6 DATA AVAILABILITY STATEMENT

We make the source code of CodeTracker and our oracle publicly available [18, 19] to enable the replication of our experiments and facilitate future research on program element tracking techniques.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful and constructive feedback to improve the work. This research was partially supported by NSERC grant RGPIN2018-05095.

REFERENCES

[1] Hassan Atwi, Bin Lin, Nikolaos Tsantalis, Yutaro Kashiwa, Yasutaka Kamei, Naoyasu Ubayashi, Gabriele Bavota, and Michele Lanza. 2021. *PyRef*. https://github.com/PyRef/PyRef

[2] Hassan Atwi, Bin Lin, Nikolaos Tsantalis, Yutaro Kashiwa, Yasutaka Kamei, Naoyasu Ubayashi, Gabriele Bavota, and Michele Lanza. 2021. PyRef: Refactoring Detection in Python Projects. In *Proceedings of the 21st IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2021)*. 136–141. https://doi.org/10.1109/SCAM52516.2021.00025

[3] Aline Brito, Marco Tulio Valente, Laerte Xavier, and Andre Hora. 2020. You broke my code: understanding the motivations for breaking changes in APIs. *Empirical Software Engineering* 25 (2020), 1458–1492. https://doi.org/10.1007/s10664-019-09756-z

[4] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. APIDiff: Detecting API breaking changes. In *IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER '18)*. IEEE Computer Society, Los Alamitos, CA, USA, 507–511. https://doi.org/10.1109/SANER.2018.8330249

[5] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. Why and how Java developers break APIs. In *IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER '18)*. IEEE Computer Society, Los Alamitos, CA, USA, 255–265. https://doi.org/10.1109/SANER.2018.8330214

[6] Mihai Codoban, Sruti Srinivasa Ragavan, Danny Dig, and Brian Bailey. 2015. Software History under the Lens: A Study on Why and How Developers Examine It. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME '15)*. IEEE Computer Society, USA, 1–10. https://doi.org/10.1109/ICSM.2015.7332446

[7] Stephen Colebourne. 2021. *Apache Commons-Lang*. https://github.com/apache/commons-lang/commit/2d06a7ce8

[8] Bradley E. Cossette and Robert J. Walker. 2012. Seeking the Ground Truth: A Retroactive Study on the Evolution and Migration of Software Libraries. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 55, 11 pages. https://doi.org/10.1145/2393596.2393661

[9] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed Hassan. 2017. A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes. *IEEE Transactions on Software Engineering* 43, 7 (July 2017), 641–657. https://doi.org/10.1109/TSE.2016.2616306

[10] Danny Dig and Ralph Johnson. 2006. How Do APIs Evolve? A Story of Refactoring. *J. Softw. Maint. Evol.* 18, 2 (March 2006), 83–107. https://doi.org/10.1002/smr.328

[11] Joakim Erdfelt. 2021. *Eclipse Jetty*. https://github.com/eclipse/jetty.project/commit/41ed9f29f

[12] Michael W. Godfrey and Lijie Zou. 2005. Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. *IEEE Transactions on Software Engineering* 31, 2 (Feb. 2005), 166–181. https://doi.org/10.1109/TSE.2005.28

[13] Sunil Govindan. 2021. *Hadoop*. https://github.com/apache/hadoop/commit/dae65f3bef8ffa34d02a37041f1dfdfeeee91845#diff-d805bcb4a1f9cddb2b92a6b5982fc41beae91419c8a1252d71af9b65f6e3d43cR1005

[14] Felix Grund, Shaiful Alam Chowdhury, Nick Bradley, Braxton Hall, and Reid Holmes. 2021. CodeShovel: Constructing Method-Level Source Code Histories. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) (ICSE '21)*. 1510–1522. https://doi.org/10.1109/ICSE43902.2021.00135

[15] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. 2011. Historage: Fine-Grained Version Control System for Java. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution* (Szeged, Hungary) *(IWPSE-EVOL '11)*. Association for Computing Machinery, New York, NY, USA, 96–100. https://doi.org/10.1145/2024445.2024463

[16] Yoshiki Higo, Shinpei Hayashi, and Shinji Kusumoto. 2020. On tracking Java methods with Git mechanisms. *Journal of Systems and Software* 165 (2020), 110571. https://doi.org/10.1016/j.jss.2020.110571

[17] Andre Hora, Danilo Silva, Marco Tulio Valente, and Romain Robbes. 2018. Assessing the Threat of Untracked Changes in Software Evolution. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 1102–1113. https://doi.org/10.1145/3180155.3180212

[18] Mehran Jodavi and Nikolaos Tsantalis. 2022. *CodeTracker*. https://doi.org/10.5281/zenodo.7080276

[19] Mehran Jodavi and Nikolaos Tsantalis. 2022. *CodeTracker source code and oracle*. https://github.com/jodavimehran/code-tracker

[20] Puneet Kapur, Bradley Cossette, and Robert J. Walker. 2010. Refactoring References for Library Migration. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. Association for Computing Machinery, New York, NY, USA, 726–738. https://doi.org/10.1145/1869459.1869518

[21] David Kawrykow and Martin P. Robillard. 2011. Non-Essential Changes in Version Histories. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 351–360. https://doi.org/10.1145/1985793.1985842

[22] Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2020. Understanding Type Changes in Java. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 629–641. https://doi.org/10.1145/3368089.3409725

[23] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. 2006. Automatic Identification of Bug-Introducing Changes. In *21st IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, Washington, DC, USA, 81–90. https://doi.org/10.1109/ASE.2006.23

[24] Zarina Kurbatova. 2021. *KotlinRMiner*. https://github.com/JetBrains-Research/kotlinRMiner

[25] Zarina Kurbatova, Vladimir Kovalenko, Ioana Savu, Bob Brockbernd, Dan Andreescu, Matei Anton, Roman Venediktov, Elena Tikhomirova, and Timofey Bryksin. 2021. RefactorInsight: Enhancing IDE Representation of Changes in Git with Refactorings Information. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE 2021)*. IEEE Computer Society, Los Alamitos, CA, USA, 1276–1280. https://doi.org/10.1109/ASE51524.2021.9678646

[26] Thomas D. LaToza and Brad A. Myers. 2010. Hard-to-Answer Questions about Code. In *Workshop on Evaluation and Usability of Programming Languages and Tools* (Reno, Nevada) *(PLATEAU '10)*. Association for Computing Machinery, New York, NY, USA, Article 8, 6 pages. https://doi.org/10.1145/1937117.1937125

[27] Yun Young Lee, Darko Marinov, and Ralph E. Johnson. 2015. Tempura: Temporal Dimension for IDEs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. 212–222. https://doi.org/10.1109/ICSE.2015.42

[28] Mehran Mahmoudi and Sarah Nadi. 2018. The Android Update Problem: An Empirical Study. In *15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. ACM, New York, NY, USA, 220–230. https://doi.org/10.1145/3196398.3196434

[29] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *IEEE International Conference on Software Maintenance (ICSM '13)*. IEEE Computer Society, Washington, DC, USA, 70–79. https://doi.org/10.1109/ICSM.2013.18

[30] Arun C Murthy, Christopher Douglas, Devaraj Das, Greg Roelofs, Jeffrey Naisbitt, Josh Wills, Jonathan Eagles, Krishna Ramachandran, Luke Lu, Mahadev Konar, Robert Evans, Sharad Agarwal, Siddharth Seth, Thomas Graves, and Vinod Kumar Vavilapalli. 2021. *Hadoop*. https://github.com/apache/hadoop/commit/dbecbe5df

[31] Jeremias Märki. 2021. *Apache Commons-IO*. https://github.com/apache/commons-io/commit/6a1bb4d53

[32] Jeremias Märki. 2021. *Apache Commons-IO*. https://github.com/apache/commons-io/commit/7748ed364

[33] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. 2013. A Comparative Study of Manual and Automated Refactorings. In *Proceedings of the 27th European Conference on Object-Oriented Programming* (Montpellier, France) *(ECOOP'13)*. Springer-Verlag, Berlin, Heidelberg, 552–576. https://doi.org/10.1007/978-3-642-39038-8_23

[34] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson. 2014. Mining Fine-Grained Code Changes to Detect Unknown Change Patterns. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 803–813. https://doi.org/10.1145/2568225.2568317

[35] Stéphane Nicoll. 2021. *Spring Framework*. https://github.com/spring-projects/spring-framework/commit/2dc674f35

[36] Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. 2021. Evaluating SZZ Implementations Through a Developer-Informed Oracle. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 436–447. https://doi.org/10.1109/ICSE43902.2021.00049

[37] Bo Shen, Wei Zhang, Haiyan Zhao, Guangtai Liang, Zhi Jin, and Qianxiang Wang. 2019. IntelliMerge: A Refactoring-aware Software Merging Technique. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 170 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360596

[38] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 858–870. https://doi.org/10.1145/2950290.2950305

[39] Danilo Silva and Marco Tulio Valente. 2017. RefDiff: Detecting Refactorings in Version Histories. In *14th International Conference on Mining Software Repositories* (Buenos Aires, Argentina) *(MSR '17)*. IEEE Press, Piscataway, NJ, USA, 269–279. https://doi.org/10.1109/MSR.2017.14

[40] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When Do Changes Induce Fixes?. In *International Workshop on Mining Software Repositories*. ACM, New York, NY, USA, 1–5. https://doi.org/10.1145/1082983.1083147

[41] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. 2014. Incremental Origin Analysis of Source Code Files. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 42–51. https://doi.org/10.1145/2597073.2597111

[42] Sunghun Kim, Kai Pan, and E. J. Whitehead. 2005. When functions change their names: automatic detection of origin relationships. In *12th Working Conference on Reverse Engineering (WCRE'05)*. 143–152. https://doi.org/10.1109/WCRE.2005.33

[43] Federico Tomassetti. 2021. *JavaParser*. https://github.com/javaparser/javaparser/commit/37f93be64

[44] Federico Tomassetti. 2021. *JavaParser*. https://github.com/javaparser/javaparser/commit/427dd53b9ebedcb0bdb687007eb0faf2de734df4#diff-d4545a64b742a6d7b072135d10643c1d309ecaa62daa9450ba05dc68985543aaR477

[45] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* 48, 3 (2022), 930–950. https://doi.org/10.1109/TSE.2020.3007722

[46] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 483–494. https://doi.org/10.1145/3180155.3180206

[47] John Verhaeg. 2021. *Hibernate ORM*. https://github.com/hibernate/hibernate-orm/commit/8c806d361

[48] Chadd Williams and Jaime Spacco. 2008. SZZ Revisited: Verifying when Changes Induce Fixes. In *Workshop on Defects in Large Software Systems* (Seattle, Washington). ACM, New York, NY, USA, 32–36. https://doi.org/10.1145/1390817.1390826

[49] Jesse Wilson. 2021. *OkHttp*. https://github.com/square/okhttp/commit/a91124b6d