Method-Level Test-to-Code Traceability Link Construction by Semantic Correlation Learning

Weifeng Sun, Zhenting Guo, Meng Yan, Zhongxin Liu, Yan Lei, Hongyu Zhang

Abstract—Test-to-code traceability links (TCTLs) establish links between test artifacts and code artifacts. These links enable developers and testers to quickly identify the specific pieces of code tested by particular test cases, thus facilitating more efficient debugging, regression testing, and maintenance activities. Various approaches, based on distinct concepts, have been proposed to establish method-level TCTLs, specifically linking unit tests to corresponding focal methods. Static methods, such as naming-convention-based methods, use heuristic- and similarity-based strategies. However, such methods face the following challenges: ① Developers, driven by specific scenarios and development requirements, may deviate from naming conventions, leading to TCTL identification failures. ② Static methods often overlook the rich semantics embedded within tests, leading to erroneous associations between tests and semantically unrelated code fragments. Although dynamic methods achieve promising results, they require the project to be compilable and the tests to be executable, limiting their usability. This limitation is significant for downstream tasks requiring massive test-code pairs, as not all projects can meet these requirements. To tackle the abovementioned limitations, we propose a novel static method-level TCTL approach, named TESTLINKER. For the first challenge of existing static approaches, TESTLINKER introduces a two-phase TCTL framework to accommodate different project types in a triage manner. As for the second challenge, we employ the *semantic correlation learning*, which learns and establishes the semantic correlations between tests and focal methods based on Pre-trained Code Models (PCMs). TESTLINKER further establishes mapping rules to accurately link the recommended function name to the concrete production function declaration. Empirical evaluation on a meticulously labeled dataset reveals that TESTLINKER significantly outperforms traditional static techniques, showing average F1-score improvements ranging from 73.48% to 202.00%. Moreover, compared to state-of-the-art dynamic methods, TESTLINKER, which only leverages static information, demonstrates comparable or even better performance, with an average F1-score increase of 37.40%.

Index Terms—Software engineering, Software testing, Traceability, Pre-trained Code Model.

1 INTRODUCTION

MAINTAINING traceability between software artifacts is crucial in the software development lifecycle [1]. Watkins *et al.* emphasize the significance of software artifact traceability, highlighting that "You can't manage what you can't trace" [2]. Unit testing validates the correctness of the unit under test (hereon *tested code*) and enhances software quality. Establishing precise traceability between test code and the tested code, known as *test-to-code traceability links* (**TCTLs**) [3], is essential for leveraging the benefits of unit testing and remains a critical part of the software development process.

Although establishing TCTLs is necessary, achieving this connection is far from straightforward, especially without explicit clues or effective tool support [4]. Moreover, substantial research has concentrated on class-level traceability, linking test classes to their corresponding focal classes [5], [6], [7], [8], [1], [9]. In contrast, method-level traceability, which pairs unit tests with specific focal methods, has not received as much attention [10], [11], [12]. The granularity provided by method-level TCTLs, as opposed to class-level TCTLs, is critical for accurately understanding and validating production functionality, as well as for effectively locating and resolving defects [12]. Therefore, this paper focuses on method-level traceability, with *test code* and *tested code* referring to the unit test (hereafter called *test*) and the focal method, respectively.

In addressing the challenge of automated traceability at the method level, researchers have proposed various static approaches, yielding promising outcomes. Some approaches use heuristics, such as naming conventions (NC), where tests are named after their focal methods, typically with a `test` prefix or suffix [13], [9], [14]. For example, a test named `testDeepCopy` likely corresponds to a focal method named `DeepCopy`. Other common approaches rely on lexical similarity between the tests and focal methods. However, these methods face effectiveness bottlenecks due to one or several of the following challenges: ① **C1**: Developers, driven by specific scenarios and development requirements, may ignore naming conventions, leading to TCTL identification failures. ② **C2**: Moreover, static methods that depend on text similarity or predefined rules often overlook the rich semantic content in test code, potentially misidentifying corresponding focal methods.

Recently, White *et al.* [3], [15] introduce TCTracer, a state-of-the-art dynamic TCTL technique. TCTracer leverages test execution information to identify functions executed by a given test t and then applies static traceability methods to score traceability links between t and each executed function. The highest-scoring production function is selected as

- *Weifeng Sun, Zhenting Guo, Meng Yan, Yan Lei, and Hongyu Zhang are with the School of Big Data and Software Engineering, Chongqing University, China.*
  E-mail: weifeng.sun@cqu.edu.cn, cqugzt@cqu.edu.cn, mengy@cqu.edu.cn, yanlei@cqu.edu.cn, hyzhang@cqu.edu.cn
- *Zhongxin Liu is with the State Key Laboratory of Blockchain and Data Security, Zhejiang University, China.*
  E-mail: liu_zx@zju.edu.cn

Meng Yan is the corresponding author.

the focal method. Unlike static TCTL methods that analyze all production functions within a project, TCTracer narrows its focus to only those executed by t, significantly reducing the number of potential function candidates. Unfortunately, its applicability reveals certain limitations: ① *Constrained Usability in Downstream Tasks*. Identifying and constructing TCTLs underpins a range of downstream tasks, such as Just-In-Time obsolete test code detection [16], co-evolution visualization [17], and assertion generation [18]. Such tasks require massive pairs of tests and the corresponding focal methods. For instance, Tufano *et al.* [19] mine a vast dataset of 887,646 test-code pairs to fine-tune a transformer model for unit test generation. Similarly, Rao *et al.* [20] prepare 1.1M test-code pairs to pre-train code models. Nevertheless, TCTracer requires the project to be compilable and the tests to be executable, restricting its usability in constructing large-scale datasets, as not all projects meet these criteria. Moreover, automating project compilation is often challenging due to dependencies on specific external or internal setups and resources [21], [22]. ② *Limitations Derived From Static TCTL Techniques.* Although TCTracer integrates dynamic information, it still relies on static TCTL strategies for scoring traceability links, inevitably inheriting limitations similar to those of static methods, particularly in challenge **C1** and **C2**.

Our work. To tackle the abovementioned challenges, we propose a more flexible static TCTL approach, with the goal of matching or surpassing the performance of dynamic methods. **For the challenge C1 mentioned above**, we introduce a two-phase TCTL framework to accommodate different project types in a triage manner. Leveraging naming conventions, our method determines focal methods for tests adhering to established conventions. For tests deviating from these conventions, we introduce an innovative identification strategy to establish TCTLs effectively. **Regarding the challenge C2**, we leverage the *semantic correlation learning*, which learns and understands the inherent semantic correlation between tests and focal methods based on semantic understanding capabilities of Pre-trained Code Models (**PCMs**). Notably, test code may involve many function calls, including the calls to helper functions and those to initialize object states. While such function calls are necessary for setting up the test environment but are not the calls to the focal method. Yet, it is fundamental for the test code to call the tested code to verify its functionality, which implies that the tested code is interwoven within the function calls invoked by the test code. The challenge lies in isolating potential function calls in the test code and understanding the intents behind these calls to identify the focal method accurately. Motivated by this, we reframe the TCTL problem as a ranking task among a set of potential function calls, aiming to identify the most relevant one as the focal method.

Based on the idea mentioned above, we propose a novel approach named TESTLINKER, which integrates heuristic rules with PCMs to facilitate the TCTL construction. Specifically, for a given test t, TESTLINKER first extracts the name of t and then applies naming conventions to determine a function name candidate for t. If this name candidate can be found within the focal class, TESTLINKER recommends the corresponding method as the focal method. Otherwise, TESTLINKER parses the test to extract the names of the functions called in the t. To filter out potential yet extraneous function calls that could muddle PCM inference, we establish two heuristic rules to refine the name list. Subsequently, TESTLINKER employs a PCM fine-tuned on our constructed dataset to evaluate the semantic correlation/relevance of each function call name to the test code, where the name with the highest relevance serves as the recommended focal method name. To map the recommended name to its function declaration, we have implemented a set of mapping rules, improving TCTL identification precision.

To evaluate the performance of TESTLINKER, we first utilize the largest publicly available dataset provided by White *et al.* [3], [15], which is manually annotated and widely recognized as a benchmark in the field. Given that this public dataset primarily consists of utility libraries, we additionally manually labeled two distinct types of projects to extend the evaluation dataset. In total, the manually labeled dataset comprises 335 verified traceability links, encompassing a diverse array of test scenarios. The results indicate that TESTLINKER significantly outperforms existing static techniques in terms of effectiveness. Specifically, TESTLINKER achieves an average precision of 73.51%, an average recall of 58.81%, and an average F1-score of 65.34%. These metrics reflect average F1-score improvements ranging from 73.48% to 202.00%. Furthermore, compared to state-of-the-art dynamic TCTL methods, TESTLINKER, which leverages only static information, demonstrates comparable or superior performance, with an average F1-score improvement of 37.40%.

Novelty & Contributions. To sum up, the contributions of this paper are as follows:

(1) **Effective Technique.** We propose TESTLINKER, a hybrid approach that integrates heuristic rules with Pre-trained Code Models (PCMs) to construct the test-to-code traceability link (TCTL), without using any additional dynamic execution information. To our knowledge, our work is the first application of PCM in this context. Moreover, we define a set of mapping rules, ensuring precise alignment of the identified focal method names with their corresponding classes and specific function details. A notable feature of TESTLINKER is its flexibility, which enables it to be implemented with various PCMs.

(2) **Extensive study.** We conduct an empirical study to evaluate the effectiveness of TESTLINKER compared to state-of-the-art static and dynamic TCTL techniques. The results indicate that TESTLINKER significantly outperforms static baselines, achieving average F1-score improvements ranging from 73.48% to 202.00%. When compared to dynamic TCTL methods, TESTLINKER consistently shows comparable or even superior effectiveness.

(3) **Open Science.** To support the research and development in the field of TCTL, we have curated and released several datasets that include tested code, test code, and the function calls within the test code, providing valuable resources for further exploration and advancement in TCTL methodologies. Additionally, we have open-sourced the scripts for data processing, model training, and model evaluation, as well as the traceability link results and related models for follow-up studies [23].

2 BACKGROUND

2.1 Test-to-Code Traceability Link

The establishment and maintenance of traceability links between test code and their corresponding tested code have gained significant attention in the research community, given their diverse applications. These traceability links are crucial for: 1) *Identifying Test Code Updates*: They help pinpoint which test code needs updates following changes in the tested code [15]. 2) *Maintaining Consistency During Refactoring*: Traceability aids in maintaining consistency during the refactoring process [24]. 3) *Documenting*: They serve as a comprehensive form of documentation [1]. 4) *Enhancing Regression Testing Efficiency*: TCTL greatly bolsters the efficiency of regression testing, leading to considerable time and resource savings [25], [26]. 5) *Facilitating Fault Localization*: Additionally, it can help developers accurately identify the specific code under test when test cases fail. 6) *Laying Foundations for Downstream Tasks*: More importantly, establishing these traceability links lays the foundation for a range of downstream tasks, such as Just-In-Time detection of obsolete test code [27], co-evolution of production and test code [17], [28], assertion generation [18], and unit test generation [19], *etc.*

The majority of previous research on TCTL has focused on the class level, *i.e.,* establishing links between test classes and their corresponding focal classes [9], [1], [7], [8], [5], [6]. For instance, Van Rompaey *et al.* [9] assess six traceability techniques in three projects, using a dataset of 59 links. Their findings suggest perfect precision and recall for naming conventions, but their study is limited to class-level traceability and a small dataset. Kicsi *et al.* [8] investigate Latent Semantic Indexing (LSI) for class-level TCTL, assuming lexical similarity between a test class and its focal class. Csuvik *et al.* [6] adopt a similar approach but use word embeddings, improving precision but not investigating recall.

Research on method-level TCTLs, which link unit tests to their focal methods, has been relatively limited [10], [12], [11]. EzUnit [10] enables developers to manually annotate tests with links to focal methods. Similarly, TestNForce [12] links tests to focal methods, employing tracing to identify methods called within a test. However, TestNForce does not perform further filtering, leading to a low precision. More recent advancements by White *et al.* [3], [15] with TCTracer mark a significant development in this area. TCTracer dynamically collects each test's function calls, including the called functions and their positions within the call stack relative to the tests. This methodology provides a natural filtering process that serves as a starting place for establishing TCTLs. TCTracer then utilizes static traceability techniques on these dynamically generated candidate links to compute scores indicating their validity. Additionally, TCTracer employs two methods—*call depth discounting* and *normalization*—to refine these scores. Call depth discounting adjusts the scores based on the call stack distance, with the intuition that functions closer to the test are more likely the focal methods. Normalization standardizes these scores, ensuring a consistent metric. Unlike the aforementioned methods, our proposed hybrid approach combines heuristic rules with Pre-trained Code Models (PCMs) to facilitate TCTL construction, without any additional dynamic execution information. Furthermore, TESTLINKER addresses the scalability and generalizability challenges of previous TCTL methods to some extent, making it adaptable to a wider range of project sizes and types.

2.2 Pre-trained Code Model

Our research draws inspiration from the success of Pre-trained Code Models (PCMs) in software engineering. Such PCMs typically fall into three categories of transformer architecture: encoder-only, decoder-only, and encoder-decoder models [29]. 1) *Encoder-Only Models*: Models like CodeBERT[30] and GraphCodeBERT [31] use a bidirectional transformer during pre-training. This allows each token in the input sequence to interact with all other tokens, providing a comprehensive understanding of the context. However, these models require an external decoder for generative tasks, which is not pre-trained and must be trained from scratch. 2) *Decoder-Only Models*: Models like GPT [32] are pre-trained using a unidirectional language modeling approach. Here, tokens can only interact with previous tokens in the sequence to predict the next tokens. This design makes them suitable for autoregressive tasks like code completion. However, the unidirectional nature limits the effectiveness of these models in understanding tasks that require a complete and integrated understanding of the entire input context. 3) *Encoder-Decoder Models*: These models combine both encoding and decoding capabilities, rendering them highly adaptable for various tasks. The encoder-decoder structure is beneficial for tasks that require an in-depth understanding of context and the generation of relevant outputs based on that context.

PCMs have been applied in a wide range of code-related tasks, including program repair [33], [34], vulnerability repair [35], [36], vulnerability detection [37], [38], code review [39], [40], and assertion generation [41], [42], [43]. For instance, Steenhoek *et al.* [38] offer a comprehensive evaluation of the latest PCMs in software vulnerability detection. Additionally, Xia *et al.* [33] assess the effectiveness of various PCMs in rectifying real-world software bugs. Recent studies [41], [42], [43] delve into the capabilities of PCMs, such as T5 and BART, in facilitating the task of assertion generation. These models undergo a process of pre-training and fine-tuning to adapt to the specific requirements. Despite promising results from PCMs in various software engineering tasks, to the best of our knowledge, there has been no research specifically investigating their capabilities in supporting TCTL. In this study, we harness PCMs to capture the testing intent of tests, thereby establishing more precise test-to-code traceability links. To explore the generalization capabilities of TESTLINKER, we have selected four advanced PCMs, *i.e.,* CodeBERT [30], GraphCodeBERT [31], UniXcoder [44], and CodeT5 [16]. Additionally, we investigate the use of CHATGPT, a large language model, to further enhance our evaluation.

3 MOTIVATION

To better illustrate our motivation, we refer to an example (as shown in Figure 1) from the popular GitHub project `commons-math` [45]. In this example, we manually establish

Test Code

```
@Test
public void testCreate() {
    final double[] d = { -1.2, 3.4, 5 };
    final DiagonalMatrix m = new DiagonalMatrix(d, false);
    final RealMatrix p = m.createMatrix(5, 5);
    final int RowDimension = p.getRowDimension();
    Assert.assertEquals(5, RowDimension);
}
```

Tested Code

```
public RealMatrix createMatrix(int rowDimension,
                               int columnDimension) {

    if (rowDimension != columnDimension) {
        throw new CustomException(rowDimension,
                                  columnDimension);
    }

    return new DiagonalMatrix(rowDimension);
}
```

Fig. 1: A test-to-code traceability link in `commons-math`

the TCTL to identify and summarize two critical challenges that impede the effectiveness of current TCTL approaches.

Challenge 1 (C1): Naming conventions in test code sometimes deviate from the norm. Although best practices for JUnit test naming suggest a resemblance between test names and focal method names, often realized by appending or prefixing `test` to the focal method name, real-world practices sometimes deviate from this norm. For instance, some tests commence with the prefix `BUG`, followed by a numeric identifier. This indicates a linkage to the specific bug report [9]. Moreover, even for tests that seem to follow naming conventions, accurately identifying their focal methods can be challenging. An example in Figure 1 demonstrates this divergence: Despite the use of the prefix `test`, the actual focal method is `createMatrix`, not `Create`. While strict adherence to naming conventions provides a precise basis for constructing TCTLs, as supported by prior research [3], [15], [9], the existence of exceptions necessitates a more flexible approach. This understanding motivates the development of our two-phase TCTL framework to accommodate different project types in a triage manner.

Challenge 2 (C2): Current static TCTL methods often overlook the semantic information within test code. Existing static methods tend to rely on naming and textual similarities, often ignoring the rich semantics embedded within test code. Test names provide limited identification information, usually consisting of just a few tokens. On the other hand, assessing text similarity between tests and production functions often fails to capture semantic nuances, as code fragments with high semantic similarity might exhibit substantial differences in lexical composition, and vice versa [46]. For instance, the test in Figure 1 only shares a Jaccard similarity of 0.175 with the focal method. The Last Call Before Assert (LCBA) technique is predicated on the assumption that the last function invoked before an assertion is the one being tested. However, this assumption is not always correct. In this instance, LCBA erroneously identifies `getRowDimension` as the focal method. Constructing precise TCTLs requires an in-depth understanding of test code semantics to capture the test's intent. In the example test, the initialization of a diagonal matrix, invocation of `createMatrix`, and subsequent assertion aim to verify the correct handling of specific inputs by `createMatrix`. This level of semantic comprehension extends beyond basic similarity measures or predefined rules. Benefiting from the pre-training paradigm, Pre-trained Code Models (PCMs) offer satisfactory semantic understanding, making them suitable for constructing TCTLs. Furthermore, we observe that tests typically call their focal methods and auxiliary functions, such as `DiagonalMatrix` and `getRowDimension`. Therefore, based on the PCMs' semantic understanding, we model the TCTL as a ranking challenge and employ the *semantic correlation learning* [47], [48], [49], which learns the inherent semantic correlation between tests and focal methods.

4 Approach

In this section, we introduce TestLinker in detail. As discussed in Section 3, to address **Challenge C1**, we propose a two-phase TCTL framework. Specifically, in cases where tests strictly adhere to naming conventions, we leverage these conventions to identify focal methods. For instances that deviate from these norms, we introduce an innovative identification strategy, ensuring the effective establishment of traceability links. To address **Challenge C2**, our identification strategy trains a neural ranking model to evaluate the semantic correlation/relevance of each called function to the test code, with a higher relevance score indicating a greater likelihood of being the focal method.

The overall framework of our approach is illustrated in Figure 2. TestLinker consists of two parts: **Offline Ranking Model Training** and **Online Two-phase TCTL Identification**. During offline training, we establish a custom training dataset to train a neural ranking model. When it comes to online construction, for a given test t, TestLinker follows these steps to identify TCTLs: 1) TestLinker first extracts the name of t and then employs the Naming Convention (NC) to identify a function name candidate for t (**refer to Component#1**). 2) If this name candidate can be found within the corresponding focal class, TestLinker recommends the corresponding method as the focal method. 3) If the name candidate is not found, TestLinker switches to its second phase (**refer to Component#2**). In this phase, TestLinker parses t to extract the names of the functions called within it, filters out irrelevant ones using heuristics, and utilizes the trained neural ranking model to identify the function name most likely to be the focal method. Given Java's polymorphic features, TestLinker implements a series of mapping rules to precisely associate the recommended function name with its corresponding production function declaration.

4.1 Offline-Ranking Model Training

Actually, a single test may evaluate multiple overloaded functions. For example, in the `gson` project [50], the `testStringValueAsSingleElementArraySerialization` tests two functions with the same name: `toJson(String[])` and `toJson(String[], String[].class)`. Besides, Java's polymorphic characteristic introduces complexity to identifying the

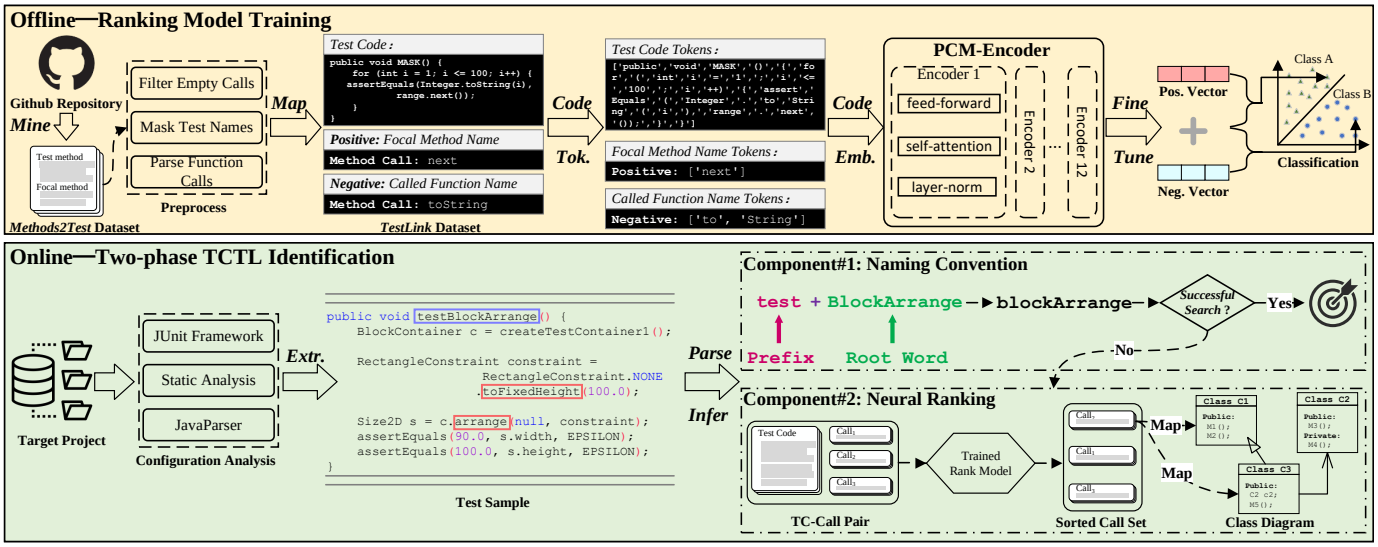

Fig. 2: The overall framework of our approach. *Offline—Ranking Model Training:* $TestLink$, an adaptation of the Methods2Test dataset, enhances model training by pairing modified developer-written tests with focal methods, resulting in over 2.5 million labeled instances for studying semantic correlations. *Online—Two-phase TCTL Identification:* Trained models are applied to a target project, ensuring unbiased validation with no overlap in data from the $TestLink$ dataset.

focal method, as the actual parameter in the test may not align with the formal reference types. This misalignment can hinder the model's ability to accurately distinguish between different function calls based on their signatures. To tackle this challenge, our ranking model prioritizes the names of functions called in a test, providing an abstract level to establish the relationship between the test and function calls. This approach is crucial, as relying on function signatures at this stage might overlook potential matches due to polymorphism-related discrepancies. When multiple calls share the same name, each is considered a potential focal method.

Our model is built using a pre-trained CodeT5 model, fine-tuned on our customized $TestLink$ dataset. It should be noted that TESTLINKER can be implemented with other PCMs (**see Section 6.4**). CodeT5 [27] adopts T5's encoder-decoder architecture and incorporates token-type information specific to code. It utilizes a denoising sequence-to-sequence pre-training task. Alongside a denoising pre-training task, it incorporates semantics from developer-assigned identifiers via two identifier-related tasks: *identifier tagging* and *masked identifier prediction*. These tasks enable CodeT5 to comprehend the significance of identifiers in code and to learn contextual dependencies involving identifiers. CodeT5 has demonstrated its utility in a wide range of downstream tasks. The $TestLink$ dataset is structured as $TestLink = \{((t,c),l)_1,\ldots,((t,c),l)_n\}$, where t signifies the test code, c denotes the function call name, and l is a binary label with $l \in \{0,1\}$. A label of 1 indicates that c is the focal point of t, while other function call names are labeled 0. More details about $TestLink$ can be found in Section 5.2.1. We model the TCTL challenge as a binary classification task. The neural ranking model accepts t and c as inputs, truncates the code tokens, and uses the CodeT5 model to embed them. After obtaining the representations for the test and the name of its invoked function, the ranking model employs a deep learning classifier to produce the final probabilities of the semantic correlation. Overall, our neural ranking model comprises an encoder and a classifier. The encoder, powered by CodeT5, generates embeddings for the input tokens, while the classifier predicts whether a function call is the focal point of the test code.

1) **Encoder:** The encoder is responsible for obtaining the contextual representative embedding of the input sequence. In this paper, we exploit the pre-trained CodeT5 encoder to initialize the encoder. Specifically, the model takes the test code paired with the name of one of its invoked functions as input, denoted as TC. We treat TC as sequences of tokens and employ a subword tokenizer [51] to address the out-of-vocabulary problem by breaking down identifiers into their subtokens. We retain the original tokenization vocabulary instead of building a new one, allowing the model to inherit the pre-trained semantic understanding and start learning prediction from a solid initial point. This token sequence is subsequently mapped to an embedding sequence $\tilde{\mathbf{X}} = \{\tilde{\mathbf{x}}_1, \tilde{\mathbf{x}}_2, \ldots, \tilde{\mathbf{x}}_n\}$ via the embedding layer. Within CodeT5, the sequence undergoes transformations across l dedicated Transformer blocks, each integrating a multi-headed self-attention mechanism [52], a feed-forward network and the layer normalization operation [53], as follows:

$$\begin{aligned} \hat{\mathbf{X}} &= \text{MultiHead}(\tilde{\mathbf{X}}) \\ \mathbf{X}^i &= \text{LayerNorm}(\hat{\mathbf{X}} + \text{FFN}(\hat{\mathbf{X}})) \end{aligned} \tag{1}$$

Here, MultiHead$(\cdot)$, FFN$(\cdot)$ and LayerNorm$(\cdot)$ represent the multi-head self-attention layer [52], the feed-forward layer and the layer normalization operation [53], respectively. i in $\mathbf{X}^i$ indicates $\mathbf{X}^i$ is the output of the i_{th} Transformer layer. The multi-head self-attention layer learns long-range dependencies in the input code tokens. The feed-forward layer linearly transforms the token embeddings for better feature extraction, and layer normalization ensures the stability of the code token embedding distribution. After being processed by l Transformer layers, TC is encoded as a sequence of contextual embeddings $\mathbf{X}^l = \{\mathbf{x}_1^l, \mathbf{x}_2^l, \ldots, \mathbf{x}_n^l\}$.

We use the last hidden state $\mathbf{x}_n^l$ as the contextual vector representation $\mathbb{R}$ of TC.

2) **Classifier:** This step involves binary classification based on the learned representation $\mathbb{R}$ to determine whether the function call name is the focal method name of the input test. To better capture the relationships between the two parts of the input ($TC_i = \{t, c\}$) from $\mathbb{R}$, we employ a dense layer with a non-linear function to learn latent interactions between them. Subsequently, the output of the dense layer is used to predict the likelihood of the final label $l = \{0, 1\}$. More precisely, the classifier is defined as follows:

$$f = \tanh(W\mathbb{R} + b)$$
$$P(l|t, c) = \text{Softmax}(f) \quad (2)$$

where W represents the weight matrix, b denotes the bias vector, and `tanh` serves as the activation function within the dense layer. The `Softmax` generates the final probability score for the label l between 0 and 1. Our objective is to train the model to recognize and assign high semantic relevance scores (*i.e.,* prediction probability) to positive samples, while negative samples are allocated low scores.

4.2 Online-Two-phase TCTL Construction

4.2.1 Key Component#1: Naming Convention

Naming Convention (NC) has been demonstrated to yield high precision in identifying relevant focal methods [15]. Although the specific conventions may differ across projects [9], the fundamental principle remains consistent: there is a recognizable pattern or relationship between the names of the test and the focal method.

As depicted in Figure 2, **Component#1** initiates its process by analyzing the name of the test. It specifically searches prefixes or suffixes (*e.g.,* `test` in the given example) to identify the key part of the test's name, which we refer to as the *root word* (*e.g.,* `blockArrange`). The root word usually corresponds to or is closely related to the name of the function or feature under test. To accommodate varied naming practices among developers, we have predefined a set of common prefix and suffix patterns, such as {`test`, `tests`, `testcase`, `testcases`}. The root word is then extracted using regular expressions. Once a potential link is identified, the next step is to verify its validity. TESTLINKER scans the production code to locate a function named with the root word. If such a function exists, it confirms the traceability link between the test and the function. To ensure the precision of **Component#1**, we also take into account the correlation between class names. Specifically, we stipulate that for a traceability link to be established in **Component#1**, both the test and production function names must align, and their respective class names must correspond.

4.2.2 Key Component#2: Neural Ranking

1) **Model Inference**: While it is straightforward to obtain function names called in the test through static analysis, applying the trained ranking model indiscriminately to all function names can result in several challenges: 1) *Computational Overhead*: Running the ranking model for each function name in a potentially large function call set can be computationally intensive due to the resources required for model inference. 2) *Compromised Prediction Precision*: The presence of function call names that are plausible but irrelevant to the test's actual focus can adversely affect prediction precision. To mitigate these issues, we introduce two heuristic rules to refine the original function call set (denoted as FC), thereby improving both identification efficiency and precision:

- **Non-Focal API Filtering**: While essential for implementing the test code, the standard JDK APIs and assertion functions are typically peripheral to the core testing focus, *i.e.,* the functions in the project under test. Therefore, our heuristic rules exclude them. Specifically, TESTLINKER leverages `JavaParser` [54] to obtain the fully qualified name (FQN) of each element e in FC. These FQNs, which encapsulate package, class, and function names, differentiate standard JDK and common assertions from application-specific calls. The differentiation is achieved through a pattern-matching mechanism using a predefined list of namespace patterns, including {`java.*`, `javax.*`, `org.junit.*`, `org.hamcrest.*`}.
- **Test Dependency Pruning**: It is common for test code to invoke other test functions due to inheritance or code reuse practices. For instance, `GraggBulirschStoerStepInterpolatorTest` in the `commons-math` project [45] calls functions from `StepInterpolatorTestUtils` for consistency validations, which should not be regarded as focal methods. To this end, TESTLINKER scans the entire project, specifically identifies functions located in the `src/test` directory, and recognizes them as part of the test suite rather than the production code. Leveraging the FQNs of these functions, TESTLINKER adeptly filters them out from the candidate set FC.

After applying these heuristic rules, we obtain a more relevant function call set, denoted as FC^*. In the inference phase, for each element $e \in FC^*$, TESTLINKER pairs the test code and the function name of e, and utilizes the trained model to evaluate the semantic correlation of the function name to the test. The name with the highest score is selected as the recommended focal method name.

2) **Function Mapping**: Following the prioritization of function call names, *i.e.,* obtaining the most likely focal method name, precise alignment between the identified function name and its corresponding production function declaration becomes imperative. Unlike dynamic methods that focus on executed functions, we analyze all functions within the project, which presents challenges, particularly when production classes contain multiple functions with identical names. For instance, a common function like `hashCode` might create linkage ambiguities, as merely using the function name could incorrectly associate a `hashCode` test with every instance of it in the project. Moreover, utilizing function signatures parsed from the test as production function declarations is imprecise due to Java's polymorphism. For example, in the `commons-io` project [55], the test case `testMoveFileToDirectory_Errors` calls the `closeQuietly` function. Parsing it with `JavaParser` reveals the parameter type as `BufferedOutputStream`, while the corresponding production code declares it as `closeQuietly(OutputStream)`. Here, the actual argument type is a subclass of the declared type. This poly-

Algorithm 1: Mapping to Function Declaration

Input: Storing the classes information of project and JDK, respectively: CIM, $JCIM$
Function-class detailed dictionary: $Func\_Detail$
The focal method name recommended by TESTLINKER: N
The test code under analysis: T
Output: Recommended tested class-method $RTCM$

1: $RTFS \leftarrow$ **GetFuncDetails(**N, T**)**
2: **for each** $m \in RTFS$ **do**
3: $params\_len \leftarrow$ **GetParaLength(**$m.parameters$**)**
4: $sig\_list \leftarrow$ **GetSignatures(**$m.class\_name, m.name, Func\_Detail$**)**
5: $sig\_list^* \leftarrow$ **FilterSignatures(**$sig\_list, params\_len$**)**
// Using mapping rule#1
6: $maxTotalScore \leftarrow 0$, $bestSig \leftarrow$ None
7: **for each** $sig_i \in sig\_list^*$ **do**
8: $totalScore \leftarrow 0$
// Using mapping rule#2
9: **for each** $param_j\_m \in m, param_j\_sig_i \in sig_i$ **do**
10: **if isMatch(**$param_j\_m, param_j\_sig_i$**) then**
11: $score = 1$
12: **else**
13: $score$ = **ObtainLcsScore(**$param_j\_m, param_j\_sig_i$**)**
14: **end if**
15: $totalScore += score$
16: **end for**
17: **if** $totalScore > maxTotalScore$ **then**
18: $maxTotalScore \leftarrow totalScore$
19: $bestSig \leftarrow sig_i$
20: **end if**
21: **end for**
22: $RTCM \leftarrow$ **Append(**$bestSig, bestSig.class\_name$**)**
23: **end for**
24: Return $RTCM$

morphism introduces challenges in accurately mapping the function call to the correct production function declaration, as the exact match of parameter types is not always straightforward. If the TCTL method does not accurately map the function call to the correct function declaration, it increases the manual review burden for developers. To mitigate these challenges, we introduce a series of mapping rules. These rules account for polymorphism and other nuances in function declarations, thereby reducing ambiguities and improving the overall precision of our method.

Algorithm 1, detailing our mapping rules, takes the following inputs: • *Classes Information of Project* (CIM): This contains inheritance dependencies among production classes within the project. • *Classes Information of Java* ($JCIM$): This details inheritance relationships among standard JDK classes, such as the hierarchy from `ArrayList` to `Object`. CIM and $JCIM$ are crucial for parameter type matching, *e.g.,* mapping `BufferedOutputStream` to `OutputStream` in the above example requires the inheritance dependency established by $JCIM$. • $Func\_Detail$: A dictionary where each key is the FQN of a function in the production code, and each value is the corresponding function signature. Its format is as follows: {`"fully.qualified.path.to.methodName": ("listOfParameters", "withTheirTypes")`}. • N: The focal method name as recommended by TESTLINKER. • T: The test code under analysis. Initially, the algorithm utilizes `JavaParser` to parse T and extract function calls that share the name N (denoted as $RTFS$), including their FQNs and actual parameter types (**Line 1**). For each function m from $RTFS$, the algorithm first counts the actual parameters of m and retrieves functions from $Func\_Detail$ with the same FQN (**Lines 3-4**). Then, it applies **mapping rule#1**: *The target function's parameter count should match that of the recommended function*. This criterion filters the initial list to sig_list^* (**Line 5**). For each function signature $sig_i \in sig\_list^*$, the algorithm uses **mapping rule#2**: *A higher parameter type similarity suggests a higher likelihood of the function being the intended target*. The algorithm compares parameter types at corresponding positions and calculates a cumulative similarity score between m and sig_i (**Lines 9-15**). The signature sig_i with the highest similarity score is then selected as the definitive mapping target for m (**Lines 17-20**). Each parameter type pair $(param_j\_m, param_j\_sig_i)$ receives a similarity score within the range $[0, 1]$, calculated as follows: 1) If $param_j\_m$ matches or is a subclass of $param_j\_sig_i$ according to CIM and $JCIM$, the score is set to 1. 2) For non-matching types, similarity scores are derived using the Longest Common Subsequence (LCS) method, as per the formula:

$$\text{Similarity Score} = \frac{|\text{LCS}(\text{param}_j\_m, \text{param}_j\_sig_i)|}{|\text{param}_j\_m|} \quad (3)$$

5 EVALUATION SETUP

This section presents our research questions, the experimental datasets, and the evaluation metrics used in the experiment.

5.1 Research Questions

We aim to answer the following research questions:

- **RQ1:** How does TESTLINKER compare in performance with established TCTL baselines?
- **RQ2:** Are all proposed strategies of TESTLINKER instrumental to its effectiveness?
- **RQ3:** How is TESTLINKER influenced by the quantity of recommended Function Names?
- **RQ4:** To what extent does the choice of code models affect the overall effectiveness of TESTLINKER?

5.2 Data Preparation

5.2.1 Training Dataset Construction

To train our neural ranking model, we introduce $TestLink$, a dataset derived from the Methods2Test dataset [19]. Methods2Test comprises developer-written unit tests paired with their focal methods, extracted from a large collection of over 91,000 open-source Java projects. The association of test cases with their respective focal methods in Methods2Test is achieved using two key rules: 1) *Name Matching*: This rule pairs a test case with a focal method by matching their names, excluding potential `Test` prefixes or suffixes. 2) *Unique Method Call*: If the intersection between the list of function calls within the test and the list of methods defined within the focal class yields a unique method, this method is regarded as the focal method. Methods2Test ensures the accuracy of these TCTLs [56] and includes 780,944 samples. Building on this precision, we design $TestLink$ to help PCMs understand the semantic correlation between the focal method and its test code. We specifically adapt the Methods2Test dataset for the TCTL task. In $TestLink$, we modify developer-written tests, including masking test names to prevent data leakage, as Methods2Test partially relies on

naming conventions. Methods2Test includes instances with empty function call lists, which are unsuitable for model training and testing. We thus exclude these instances from Methods2Test. For each filtered sample $\langle f_i, t_i \rangle$ from the Methods2Test, we parse t_i to obtain the list of invoked function names: $\{c_{i1}, c_{i2}, \ldots, c_{in}\}$. We then automate the labeling process based on whether the function call name exactly matches the focal method name. Specifically, if c_{im} matches the function name of f_i, we construct a positive sample $\langle t_i, c_{im}, '1' \rangle$. The remaining calls are used to construct negative samples: $\{\langle t_i, c_{ik}, '0' \rangle \mid 1 \leq k \leq n, k \neq m\}$. This construction method generates at least two new samples (one positive and multiple negative samples) for each entry in the refined Methods2Test dataset. As a result, the $TestLink$ dataset contains over 2.5 million labeled instances, making it a substantial resource for developing and evaluating TCTL prediction models.

5.2.2 Evaluation Dataset Construction

For **RQ1-RQ3**, we use the largest manually labeled dataset to the best of our knowledge, based on the following considerations: 1) **Real-World Representation**: The manually labeled dataset includes diverse types of tests, accurately representing real-world scenarios. This diversity makes it ideal for evaluating the effectiveness of various TCTL methods. 2) **Dynamic Method Requirements**: Dynamic TCTL methods require execution information, which cannot be derived from the static $TestLink$ dataset. The manually labeled dataset provides such information by recovering the project's compilation environment and containing test execution data. Below, we provide a detailed description of this manually labeled dataset.

Public Dataset. We adopt the method-level traceability datasets provided by White *et al.* [3], [15], derived from four well-known open-source Java projects that use the JUnit testing framework: Commons IO, Commons Lang, JFreeChart, and Gson. We diligently adhere to the previous experimental setting [15], aligning with the evaluated subjects and versions, restoring compilation environments, and ensuring test executability. These datasets are widely used due to their rigorous annotation process, which is conducted by a doctoral student experienced in software traceability and two senior undergraduate students.

Additional Evaluation Dataset. To further enhance the robustness and applicability of our evaluation, we have broadened the evaluation dataset by incorporating additional projects. we select these projects using the following criteria: 1) *Project Types*. The project needs to cover different types and application scenarios to ensure a diverse dataset. 2) *Long change history*. Following previous work [57], [58], [28], we exclude projects with a commit history of fewer than three years to ensure that the selected projects are well-maintained. 3) *Popularity*. The number of stars [59] on GitHub reflects a repository's popularity. Following previous research [60], we select projects with more than ten stars to avoid possible toy projects. 4) *Compilability*. The selected projects must be compilable, and all tests can be run successfully, allowing us to obtain test execution information. Given the time cost of manual labelling and the complexity of restoring the project's compilation environment, we ultimately chose Jenkins [61] and Dubbo [62]. Jenkins is a widely used automation server in the DevOps domain, which offers a rich set of plugins to support building, deploying, and automating any project. Dubbo is a high-performance Java-based open-source Remote Procedure Call (RPC) framework that provides functionalities such as automatic service registration and discovery, load balancing, and fault tolerance.

To establish method-level *oracle links*, we invite three volunteers experienced in TCTL research, each with at least four years of Java programming experience, including one PhD student and two master's students. *Oracle links*, also known as *ground truth*, refer to the manually verified traceability links that serve as the benchmark for evaluating the performance of TCTL methods. Following the methodology outlined in previous work [3], [15], each volunteer independently reviews a random selection of tests from the projects to determine the focal methods for each test. The volunteers are guided to label each sample based on several criteria: the functions called within each test, the frequency of those calls, the frequency with which these called functions are invoked by other tests, the names of the tests, and the functions whose return values are checked by assertions. After completing their individual assessments, the three volunteers collectively review any cases of disagreement. Through this collaborative review process, they reach a final consensus, achieving a full inter-rater agreement. By involving multiple reviewers and employing a rigorous review process, we ensure the reliability and accuracy of the constructed oracle links.

In total, the manually labeled dataset, incorporating both the public dataset and the additional evaluation dataset, comprises 335 manually verified traceability links. The dataset includes 231 samples across six projects: four utility libraries for different application scenarios, an automation server project, and an open-source remote procedure call (RPC) framework. An in-depth inspection reveals that the number of unique function calls per test ranges from 1 to 26, with a median ranging from 3 to 10. Here, *unique function calls* refer to the different functions invoked within a test case, considering each function only once regardless of the number of times it is called within the test. The total number of assertions ranges from 68 in `Commons IO` to 296 in `Commons Lang`, with a median ranging from 1 to 3 per test across projects. The count of assertions is determined by the number of assertion statements (e.g., `assertTrue`, `assertEquals`) found within the test code. This diversity reflects the complex nature of real-world projects. Specifically, 19.91% of samples contain multiple focal methods, and 9.96% of the tests involve indirect focal method calls. Notably, some samples include multiple traceability links due to the presence of tests with multiple focal methods. This explains why the number of traceability links (335) exceeds the number of samples (231). To avoid data leakage, we carefully remove all samples used to build the manually labeled dataset from our $TestLink$. Further details about these subjects are available in Table 1.

For **RQ4**, we utilize both the manually labeled dataset and our self-constructed $TestLink$ dataset to evaluate the performance of different PCMs in TCTL prediction. The $TestLink$ dataset is a static dataset designed specifically for training and evaluating semantic associations between

TABLE 1: Subject statistics

Project	Version	Num. of Functions	Num. of Tests	Total Num. of Assertion[1]	Median Num. of Assertion[1]	Total Num. of Unique Function Call[2]	Median Num. of Unique Function Call[2]	Num. of Samples	Num. of Ground Truth Links
Commons IO	2.5	1,246	994	68	2	186	7	28	41
Commons Lang	3.7	3,111	3,061	296	3	242	6	42	78
JFreeChart	1.0.19	9,053	2,244	110	3	208	8	28	44
Gson	2.8.0	635	1,006	107	1	215	3	50	55
Jenkins-core	2.346.1	11,476	928	149	3	267	5	42	50
Dubbo-rpc	3.2.12	957	136	124	2	445	10	41	67

1. The "Total Num. of Assertions" and "Median Num. of Assertions" columns refer to the count of assertion statements within the test code.
2. The "Total Num. of Unique Function Calls" and "Median Num. of Unique Function Calls" columns refer to the count of different functions called within the test code, considering each function only once regardless of the number of calls.

test code and focal methods. This dataset is valuable for fine-tuning and testing PCMs due to its size and specific focus on semantic relationships. Despite its simplicity and constraints—such as ensuring each test has only one corresponding focal method and requiring direct calls—the $TestLink$ dataset provides a controlled environment to evaluate PCM performance. The inclusion of $TestLink$ in RQ4 aims to broaden the evaluation scope, allowing for a more comprehensive comparison of various PCMs. By leveraging the larger and more focused $TestLink$ dataset alongside the manually labeled dataset, we can systematically investigate and compare the performance of different models, ultimately identifying the most effective model configurations for TCTL prediction. Notably, the division of the $TestLink$ dataset into training, validation, and test sets strictly aligns with the same partitioning as the Methods2Test dataset, ensuring that there is no overlap between the training and test sets in $TestLink$.

5.3 Evaluation Metrics

We have selected three critical metrics for our evaluation: precision, recall, and the F1-score. Precision and recall, which are essential in assessing binary classifier performance, measure the proportion of true positives among all positive predictions (*i.e.*, $Precision = \frac{TP}{TP+FP}$) and the proportion of all positive samples retrieved (*i.e.*, $Recall = \frac{TP}{TP+FN}$), respectively. In this context, TP represents the correctly identified focal methods by TESTLINKER, FP represents the function calls mistakenly classified as focal methods, and FN accounts for focal methods that TESTLINKER fails to recognize. As precision and recall often involve trade-offs, the F1-score (*i.e.*, $F1\text{-}score = 2 * \frac{Precision*Recall}{Precision+Recall}$) serves as a valuable metric. It quantifies the balance between precision and recall by taking their harmonic mean. Importantly, when computing precision and recall, we consider all the correct focal methods of a given test. In other words, our precision, recall, and F1-score calculations are based on the number of traceability links, not the number of samples.

5.4 Test environment

We implement TESTLINKER in Python using PyTorch [63]. The experiments are performed on a Ubuntu 20.04 with four NVIDIA GeForce RTX 3090 GPUs, one 32-core processor, and 256GB memory. We set the number of recommended function names (denoted as m) to one. In RQ3, we explore the effects of varying m on the performance of TESTLINKER. In RQ4, for other PCMs training, we take as input the tests and the names of their invoked functions, denoted as TC. PCMs tokenize and embed TC. The representation of TC is derived from either the mean or `[CLS]` vector from the last hidden states. A dense layer with a non-linear activation function predicts the labels. Inputs exceeding 512 tokens are truncated. The AdamW [64] optimizes the model, with the best version chosen by the highest harmonic mean of F1-score and accuracy on the validation dataset.

6 RESULT ANALYSIS

6.1 RQ1: Comparison with State-of-the-arts

6.1.1 Baselines

In this section, we evaluate TESTLINKER against 16 TCTL techniques, encompassing both static and dynamic strategies. Due to space limitations, we offer brief overviews of these baselines here. More detailed method descriptions and example listings for each technique can be found in the **Appendix** (see supplementary material).

Static Techniques: Static methods for establishing TCTLs solely rely on static information from the production code of the target project. They include: **1)** ***Static Naming Conventions (Static NC)*** **[9]:** This method associates a test with a production function if both their names and their respective class names match according to naming conventions. **2)** ***Static Naming Conventions – Contains (Static NCC)*** ***[3], [15]*****:** This approach establishes a link between a test and a production function when the test's name includes the function's name, and the test class's name contains the function class's name, with `test` removed from both names. **3)** ***Static Longest Common Subsequence – Both (Static LCS-B)*** ***[3], [15]*****:** This method uses Fully Qualified Names (FQNs) to determine the ratio of the longest common subsequence's length to the length of the longer entity (either the production function or test name). A higher ratio indicates a higher likelihood of the function being tested. **4)** ***Static Longest Common Subsequence – Unit (Static LCS-U)*** ***[3], [15]*****:** This method is similar to *Static LCS-B*, but the length of the longest common subsequence is divided by the length of the function's FQN only. **5)** ***Static Levenshtein Distance (Static Leven)*** ***[3], [15]*****:** This measures the edit distance between the function and test FQNs. The pair with the smallest Levenshtein distance is considered most likely linked. **6)** ***Static Last Call Before Assert (Static LCBA)*** ***[65], [9]*****:** This method operates an assumption that the function called immediately before an assert is likely to be the one being tested. **7)** ***Static Term Frequency–Inverse Document Frequency (Static TFIDF)*** ***[66]*****:** This method treats tests as documents and functions as terms, aiming to link them based on the frequency of function execution within tests.

Dynamic Techniques. TCTracer [3], [15] is the state-of-the-art dynamic TCTL framework. Consequently, this paper establishes several dynamic baselines based on the TCTracer. Leveraging the TCTracer framework, we adapt the abovementioned static baselines into dynamic variants, specifically: **1)** ***Naming Convention (TCTracer$_{NC}$)*** [3], [15], **2)** ***Naming Conventions – Contains (TCTracer$_{NCC}$)*** [3], [15], **3)** ***Longest Common Subsequence – Both (TCTracer$_{LCS\text{-}B}$)*** [3], [15], **4)** ***Longest Common Subsequence – Unit (TCTracer$_{LCS\text{-}U}$)*** [3], [15], **5)** ***Levenshtein Distance (TCTracer$_{Leven}$)*** [3], [15], **6)** ***Last Call Before Assert (TCTracer$_{LCBA}$)*** [3], [15], and **7)** ***Term Frequency–Inverse Document Frequency (TCTracer$_{TFIDF}$)*** [3], [15]. Additionally, an extra baseline, **8)** ***TCTracer$_{Tarantula}$*** [3], [15], is introduced. Drawing inspiration from Tarantula [67], an automatic fault localization technique, this baseline calculates the suspiciousness of a function relative to a test for a specific test-to-code pair where the test executes the function. Based on the intuition that each technique can provide insights not fully accessible through any other method, White *e.g.,* [3], [15] ultimately propose an integrated approach, ***TCTracer$_{Comb}$***, that combines the individual techniques described above into a single, comprehensive score. All dynamic techniques utilize dynamic trace information, allowing them to avoid common issues associated with static techniques, such as over-approximation and the inability to reason about references and dependencies resolved at runtime.

Each technique leverages different aspects of the relationship between tests and code to establish TCTLs. However, certain methods are excluded from our baselines due to specific limitations. 1) **Fixture Element Types (FET)**: This method utilizes test fixture elements (*e.g.,* objects, data) to establish traceability links. However, it is not suitable for method-level application due to its focus on broader fixture elements rather than specific methods [9]. Additionally, this strategy falls short if no objects are declared as explicit fixture elements. 2) **SCOTCH+**: This technique enhances traditional traceability by considering both dynamic slicing and textual analysis. This method identifies tested class candidates through dynamic slicing of execution traces, focusing on the last executed assert statement. It then refines this candidate set using textual similarity measures. SCOTCH+ is not well-suited for method-level TCTL because its granularity and class-level focus do not capture the finer details required for method-level traceability [1]. 3) **Lexical Analysis (LA)**: This method relies on the simple lexical similarity between test and code elements, such as natural language used in type names, identifiers, strings, and comments, to establish links. It is excluded due to its low precision and recall in practical scenarios, as confirmed by previous studies [68], [8]. 4) **Co-Evolution (Co-Ev)**: This technique analyzes the co-evolution of tests and production code over time to infer traceability links. Co-Ev is not included in our evaluation because it necessitates extensive historical co-evolution data. Furthermore, previous studies have shown that Co-Ev often suffers from low precision and recall [68], [8].

To ensure a direct and fair comparison between TESTLINKER and the baselines, when multiple functions achieve the same maximum traceability link score, the baseline randomly selects one as the focal method for that test. Given the variability introduced by this randomness, each experiment is repeated five times per baseline. We report the average results of these repetitions as the final baseline performance to ensure reliable and consistent evaluation.

6.1.2 Results

In this RQ, we construct and utilize a large-scale, manually labeled dataset. This dataset contains 335 oracle links, which are manually verified traceability links serving as ground truth for evaluating TCTL methods. The dataset spans various project types, including four utility libraries for different application scenarios, an automation server project, and an open-source remote procedure call (RPC) framework. It features a diverse array of test scenarios, encompassing different test naming conventions, varying numbers of focal methods per test, and both direct and indirect function calls to focal methods. This diversity ensures a realistic and comprehensive evaluation of TCTL methods. Further details about the dataset are provided in Section 5.2.2.

The experimental results, as shown in Tables 2 and 3, highlight the performance of TESTLINKER compared to various dynamic-based and static-based baseline techniques. The evaluation metrics considered are precision, recall, and F1-score. It is important to note that TESTLINKER is a static method. When juxtaposed with other static TCTL techniques, TESTLINKER consistently outperforms nearly all baselines, as evidenced in Table 2. While TESTLINKER falls short in precision compared to *Static NC*, it excels across almost all other evaluated metrics. This is particularly evident in its F1-score, where TESTLINKER shows significant improvements over static baselines, with average increments ranging from 73.48% to 202.00%. Such findings highlight TESTLINKER's effectiveness in static TCTL contexts. In comparison with dynamic methods, TESTLINKER exhibits a balanced performance overall. In contrast to methods like TCTracer$_{NC}$, which achieves high precision but low recall, or TCTracer$_{TFIDF}$, known for its strong recall but lower precision, TESTLINKER effectively strikes a balance in these two metrics. This is exemplified by its average F1-score of 65.34%, which reflects substantial improvements over almost all dynamic techniques, with average enhancements varying from 8.57% to 161.36%. The average F1-score performance gap between TESTLINKER and TCTracer$_{Comb}$ is marginal, at just 1.64%. Importantly, TESTLINKER achieves similar or even better effectiveness compared to dynamic methods, using only static information. TESTLINKER circumvents the need for compiling projects and executing tests, making TESTLINKER especially suitable for downstream tasks requiring massive test-code pairs.

We further conduct statistical analysis to determine whether TESTLINKER significantly outperforms all baseline methods by employing the Wilcoxon signed-rank test [69] at a 95% significance level. Meanwhile, the non-parametric Cliff's delta effect size is used to evaluate the magnitude of the difference[1] between the two approaches. Specifically, we compute the p-value of TESTLINKER with respect to each baseline, based on the F1-score results for the six

1. We use the following mapping for the values of the delta that are less than 0.147, between 0.147 and 0.33, between 0.33 and 0.474 and above 0.474 as "*Negligible (N)*", "*Small (S)*", "*Medium (M)*", "*Large (L)*" effect size, respectively [70]

TABLE 2: Effectiveness (%) of TESTLINKER compared with each baseline

Project	Category	Technique	Prec.	Recall	F1-score	Project	Category	Technique	Prec.	Recall	F1-score
Commons Lang	Dynamic-based	$\text{TCTracer}_{\text{Comb}}$	88.10	47.44	61.67 ↑12.61%	Commons IO	Dynamic-based	$\text{TCTracer}_{\text{Comb}}$	75.00	51.22	60.87 ↓-7.59%
		$\text{TCTracer}_{\text{NC}}$	100.00	10.26	18.60 ↑273.24%			$\text{TCTracer}_{\text{NC}}$	100.00	7.32	13.64 ↑312.50%
		$\text{TCTracer}_{\text{NCC}}$	96.30	33.33	49.52 ↑40.22%			$\text{TCTracer}_{\text{NCC}}$	100.00	39.02	56.14 ↑0.20%
		$\text{TCTracer}_{\text{LCS-U}}$	78.57	42.31	55.00 ↑26.25%			$\text{TCTracer}_{\text{LCS-U}}$	67.86	46.34	55.07 ↑2.14%
		$\text{TCTracer}_{\text{LCS-B}}$	73.81	39.74	51.67 ↑34.40%			$\text{TCTracer}_{\text{LCS-B}}$	64.29	43.90	52.17 ↑7.81%
		$\text{TCTracer}_{\text{Leven}}$	82.93	43.59	57.14 ↑21.52%			$\text{TCTracer}_{\text{Leven}}$	75.00	51.22	60.87 ↓-7.59%
		$\text{TCTracer}_{\text{LCBA}}$	81.08	38.46	52.17 ↑33.09%			$\text{TCTracer}_{\text{LCBA}}$	45.83	26.83	33.85 ↑66.19%
		$\text{TCTracer}_{\text{Tarantula}}$	78.57	42.31	55.00 ↑26.25%			$\text{TCTracer}_{\text{Tarantula}}$	71.43	48.78	57.97 ↓-2.97%
		$\text{TCTracer}_{\text{TFIDF}}$	88.10	47.44	61.67 ↑12.61%			$\text{TCTracer}_{\text{TFIDF}}$	75.00	51.22	60.87 ↓-7.59%
	Static-based	Static NC	90.00	11.54	20.45 ↑239.48%		Static-based	Static NC	100.00	7.32	13.64 ↑312.50%
		Static NCC	63.33	24.36	35.19 ↑97.36%			Static NCC	43.75	17.07	24.56 ↑129.02%
		Static LCS-U	54.05	25.64	34.78 ↑99.64%			Static LCS-U	34.62	21.95	26.87 ↑109.38%
		Static LCS-B	45.95	21.79	29.57 ↑134.87%			Static LCS-B	47.62	24.39	32.26 ↑74.38%
		Static Leven	51.35	24.36	33.04 ↑110.15%			Static Leven	42.86	21.95	29.03 ↑93.75%
		Static LCBA	78.79	33.33	46.85 ↑48.22%			Static LCBA	43.75	17.07	24.56 ↑129.02%
		Static TFIDF	78.05	41.03	53.78 ↑29.12%			Static TFIDF	53.57	36.59	43.48 ↑29.38%
		TESTLINKER	75.76	64.10	69.44 ——			TESTLINKER	78.26	43.90	56.25 ——
Gson	Dynamic-based	$\text{TCTracer}_{\text{Comb}}$	84.00	76.36	80.00 ↑6.48%	JFreeChart	Dynamic-based	$\text{TCTracer}_{\text{Comb}}$	92.86	59.09	72.22 ↓-20.34%
		$\text{TCTracer}_{\text{NC}}$	100.00	9.09	16.67 ↑411.08%			$\text{TCTracer}_{\text{NC}}$	100.00	15.91	27.45 ↑109.57%
		$\text{TCTracer}_{\text{NCC}}$	84.62	20.00	32.35 ↑163.28%			$\text{TCTracer}_{\text{NCC}}$	100.00	25.00	40.00 ↑43.83%
		$\text{TCTracer}_{\text{LCS-U}}$	72.00	65.45	68.57 ↑24.22%			$\text{TCTracer}_{\text{LCS-U}}$	84.62	50.00	62.86 ↓-8.47%
		$\text{TCTracer}_{\text{LCS-B}}$	70.00	63.64	66.67 ↑27.77%			$\text{TCTracer}_{\text{LCS-B}}$	80.77	47.73	60.00 ↓-4.12%
		$\text{TCTracer}_{\text{Leven}}$	78.00	70.91	74.29 ↑14.67%			$\text{TCTracer}_{\text{Leven}}$	88.46	52.27	65.71 ↓-12.45%
		$\text{TCTracer}_{\text{LCBA}}$	70.45	56.36	62.63 ↑36.01%			$\text{TCTracer}_{\text{LCBA}}$	73.08	43.18	54.29 ↑5.98%
		$\text{TCTracer}_{\text{Tarantula}}$	68.00	61.82	64.76 ↑31.53%			$\text{TCTracer}_{\text{Tarantula}}$	74.07	45.45	56.34 ↑2.12%
		$\text{TCTracer}_{\text{TFIDF}}$	70.00	63.64	66.67 ↑27.77%			$\text{TCTracer}_{\text{TFIDF}}$	62.96	38.64	47.89 ↑20.14%
	Static-based	Static NC	100.00	5.45	10.34 ↑723.41%		Static-based	Static NC	75.00	6.82	12.50 ↑360.24%
		Static NCC	63.64	12.73	21.21 ↑301.56%			Static NCC	44.44	9.09	15.09 ↑281.14%
		Static LCS-U	38.89	25.45	30.77 ↑176.84%			Static LCS-U	47.83	25.00	32.84 ↑75.21%
		Static LCS-B	21.95	16.36	18.75 ↑354.29%			Static LCS-B	56.52	29.55	38.81 ↑48.25%
		Static Leven	33.33	23.64	27.66 ↑207.96%			Static Leven	68.18	34.09	45.45 ↑26.57%
		Static LCBA	33.33	16.36	21.95 ↑288.06%			Static LCBA	50.00	13.64	21.43 ↑168.47%
		Static TFIDF	18.00	16.36	17.14 ↑396.97%			Static TFIDF	39.29	25.00	30.56 ↑88.28%
		TESTLINKER	86.79	83.64	85.18 ——			TESTLINKER	72.41	47.73	57.53 ——
Jenkins	Dynamic-based	$\text{TCTracer}_{\text{Comb}}$	76.19	64.00	69.57 ↓-12.24%	Dubbo	Dynamic-based	$\text{TCTracer}_{\text{Comb}}$	73.17	44.78	55.56 ↓-0.17%
		$\text{TCTracer}_{\text{NC}}$	100.00	20.00	33.33 ↑83.16%			$\text{TCTracer}_{\text{NC}}$	93.75	22.39	36.14 ↑53.45%
		$\text{TCTracer}_{\text{NCC}}$	88.89	32.00	47.06 ↑29.74%			$\text{TCTracer}_{\text{NCC}}$	92.59	37.31	53.19 ↑4.27%
		$\text{TCTracer}_{\text{LCS-U}}$	63.16	48.00	54.55 ↑11.93%			$\text{TCTracer}_{\text{LCS-U}}$	75.68	41.79	53.85 ↑3.00%
		$\text{TCTracer}_{\text{LCS-B}}$	52.63	40.00	45.45 ↑34.32%			$\text{TCTracer}_{\text{LCS-B}}$	54.05	29.85	38.46 ↑44.20%
		$\text{TCTracer}_{\text{Leven}}$	63.16	48.00	54.55 ↑11.93%			$\text{TCTracer}_{\text{Leven}}$	70.27	38.81	50.00 ↑10.92%
		$\text{TCTracer}_{\text{LCBA}}$	46.88	30.00	36.59 ↑66.88%			$\text{TCTracer}_{\text{LCBA}}$	44.12	22.39	29.70 ↑86.72%
		$\text{TCTracer}_{\text{Tarantula}}$	44.74	34.00	38.64 ↑58.02%			$\text{TCTracer}_{\text{Tarantula}}$	50.00	28.36	36.19 ↑53.25%
		$\text{TCTracer}_{\text{TFIDF}}$	55.26	42.00	47.73 ↑27.92%			$\text{TCTracer}_{\text{TFIDF}}$	52.63	29.85	38.10 ↑45.59%
	Static-based	Static NC	100.00	22.00	36.07 ↑69.28%		Static-based	Static NC	92.31	17.91	30.00 ↑84.87%
		Static NCC	63.64	28.00	38.89 ↑56.99%			Static NCC	84.00	31.34	45.65 ↑21.49%
		Static LCS-U	59.46	44.00	50.57 ↑20.72%			Static LCS-U	64.86	35.82	46.15 ↑20.17%
		Static LCS-B	40.54	30.00	34.48 ↑77.05%			Static LCS-B	44.44	23.88	31.07 ↑78.52%
		Static Leven	48.78	40.00	43.96 ↑38.89%			Static Leven	59.46	32.84	42.31 ↑31.09%
		Static LCBA	48.15	26.00	33.77 ↑80.81%			Static LCBA	53.85	20.90	30.11 ↑84.21%
		Static TFIDF	37.50	30.00	33.33 ↑83.16%			Static TFIDF	31.71	19.40	24.07 ↑130.38%
		TESTLINKER	64.44	58.00	61.05 ——			TESTLINKER	63.46	49.25	55.46 ——

↑ denotes performance improvement of TESTLINKER against state-of-the-art baselines
↓ denotes performance decrease of TESTLINKER against state-of-the-art baselines

evaluation projects. The statistical analysis, provided in Table 3, underscores the robustness of TESTLINKER's improvements. The p-values indicate that the F1-score improvements are statistically significant ($p < 0.05$) in most cases. The effect size measurements indicate substantial practical significance, with TESTLINKER achieving large effect sizes (L) against 12 baseline techniques. Although the comparisons with $\text{TCTracer}_{\text{LCS-U}}$ and $\text{TCTracer}_{\text{Leven}}$ are not statistically significant, TESTLINKER still demonstrates more than an 8% improvement in average F1-score. Similarly, while TESTLINKER's F1-score is slightly lower than that of $\text{TCTracer}_{\text{Comb}}$, the statistical analysis shows that this difference is not significant. This suggests that TESTLINKER's performance is comparable to the state-of-the-art dynamic TCTL technique.

Detailed Analysis with an End-to-End Example. The depicted example (Figure 3) showcases a testing scenario from the `Commons IO` project. In this example, the method

Test Code

```
@Test
public void testIO300() throws Exception {
    final File testDirectory = getTestDirectory();
    final File src = new File(testDirectory, "dir1");
    final File dest = new File(src, "dir2");
    try {
        FileUtils.moveDirectoryToDirectory(src, dest, false);
        fail("expected IOException");
    } catch (final IOException ioe) {
        // expected
    }
    assertTrue(src.exists());
}
```

Fig. 3: A test sample that is successfully handled by TESTLINKER in `Commons-IO` project

`testIO300` tests the functionality of the `FileUtils` class, specifically the `moveDirectoryToDirectory` method. Static methods such as NC, NCC, LCS-U, LSC-B, and

TABLE 3: Average Results, P-Values, Cliff's Delta and Effect Size Comparing F1-score for Our Approach with Baselines

Category	Technique	Metrics				Statistical Results		
		Prec.	Recall	F1-score		p-value	Cliff's Delta	Effect-Size
Dynamic-based	$\text{TCTracer}_{\text{Comb}}$	81.39	56.12	66.43	↓ -1.64%	-	-	-
	$\text{TCTracer}_{\text{NC}}$	97.96	14.33	25.00	↑ 161.36%	*	1.00	*L*
	$\text{TCTracer}_{\text{NCC}}$	93.75	31.34	46.98	↑ 39.08%	*	0.94	*L*
	$\text{TCTracer}_{\text{LCS-U}}$	73.30	48.36	58.27	↑ 12.13%	-	-	-
	$\text{TCTracer}_{\text{LCS-B}}$	65.61	43.28	52.16	↑ 25.27%	*	0.61	*L*
	$\text{TCTracer}_{\text{Leven}}$	75.91	49.85	60.18	↑ 8.57%	-	-	-
	$\text{TCTracer}_{\text{LCBA}}$	61.42	36.12	45.49	↑ 43.64%	*	0.78	*L*
	$\text{TCTracer}_{\text{Tarantula}}$	64.13	42.69	51.25	↑ 27.48%	*	0.50	*L*
	$\text{TCTracer}_{\text{TFIDF}}$	67.71	45.07	54.12	↑ 20.73%	*	0.39	*M*
Static-based	Static NC	93.18	12.24	21.64	↑ 202.00%	*	1.00	*L*
	Static NCC	63.72	21.49	32.14	↑ 103.28%	*	1.00	*L*
	Static LCS-U	51.02	29.85	37.66	↑ 73.48%	*	1.00	*L*
	Static LCS-B	41.03	23.88	30.19	↑ 116.44%	*	1.00	*L*
	Static Leven	49.75	29.25	36.84	↑ 77.35%	*	1.00	*L*
	Static LCBA	53.19	22.39	31.51	↑ 107.35%	*	1.00	*L*
	Static TFIDF	41.67	28.36	33.75	↑ 93.61%	*	1.00	*L*
	TESTLINKER	73.51	58.81	65.34	——	——	——	——

1. $* \; p < 0.05$, $- \; p > 0.05$.
2. *L*, *M*, *S* and *N* represent Large, Medium, Small and Negligible effect size according to Cliff's delta.

Leven encounter difficulties in establishing TCTL due to the test being generically named, resulting in zero similarity scores. Dynamic methods, on the other hand, analyze the function calls within the test execution trace. Although they perform better by narrowing down potential function candidates, they still suffer from false negatives and false positives due to their reliance on static similarity scores and execution context. In contrast, our proposed approach, TESTLINKER, extracts function calls directly from the test code and uses a neural ranking model to learn and establish semantic correlations between tests and focal methods from existing test-code pairs. The detailed steps are as follows: 1) **Component Selection**: Since TESTLINKER cannot identify the focal method for this test using naming conventions, it switches to the **Component#2**. 2) **Function Call Extraction**: TESTLINKER employs the `JavaParser` to extract the following function calls from the test to form a function call set: {`getTestDirectory`, `org.apache.commons.io.File`, `org.apache.commons.io.FileUtils.moveDirectoryToDirectory`, `org.junit.Assert.fail`, `org.junit.Assert.assertTrue`, `java.io.File.exists`}. 3) **Function Call Set Filtering**: Since `getTestDirectory` is a private function of the test class, TESTLINKER removes it from the function call set according to the **Test Dependency Pruning** guidelines. Additionally, {`org.junit.Assert.fail`, `org.junit.Assert.assertTrue`, and `java.io.File.exists`} belong to JUnit and Java standard APIs, and are also removed according to the **Non-Focal API Filtering** principle. The final set of function calls contains only {`org.apache.commons.io.File`, `org.apache.commons.io.FileUtils.moveDirectoryToDirectory`}. 4) **Neural Ranking Model**: TESTLINKER then applies its neural ranking model to evaluate the filtered function call set in the context of the test code. The model, trained on a large corpus of test-code pairs, incorporates both syntactic and semantic analysis to rank the likelihood of each function being the focal method. Based on this evaluation, TESTLINKER identifies `moveDirectoryToDirectory` as the primary focal method name with high confidence. 5) **Function Mapping**: Finally, TESTLINKER uses the information parsed by `JavaParser`, such as the number and type of parameters, to match the function declarations in the production code one by one, as described in Algorithm 1. This process correctly identifies the function declaration of the focal method: `moveDirectoryToDirectory(File, File, boolean)`.

Answer to RQ1: Overall, our comparison results reveal that, (1) TESTLINKER can achieve remarkable performance compared to existing static-based techniques with average F1-score improvements ranging from 73.48% to 202.00%; (2) TESTLINKER can achieve similar or even better effectiveness using only static information, compared to dynamic methods, with an average F1-score improvement of 37.40%.

6.2 RQ2: Ablation Experiment

6.2.1 Baselines

RQ2 aims to assess the individual contributions of the distinct components within TESTLINKER. We consider how each phase impacts TESTLINKER's effectiveness by comparing the TESTLINKER against two of its variants: 1) TESTLINKER-w/o C1: This variant constructs TCTLs solely using the neural ranking model. 2) TESTLINKER-w/o C2: Conversely, this variant relies exclusively on naming conventions. Additionally, to understand the significance of the filtering rules, which refine the list of function calls during model inference, we introduce another variant: 3) TESTLINKER-w/o rules: This variant still utilizes the two-phase framework but does not apply the filtering rules during the inference stage, meaning it evaluates all function call names in the test code.

TABLE 4: The overall effectiveness of each component of TESTLINKER

Variants	Pre.	Recall	F1-score
TESTLINKER-w/o C1	73.31%	58.21%	64.89%
TESTLINKER-w/o C2	93.18%	12.24%	21.64%
TESTLINKER-w/o rules	67.84%	57.31%	62.14%
TESTLINKER	**73.51%**	**58.81%**	**65.34%**

6.2.2 Results

Due to space limitations, Table 4 presents the average performance of TESTLINKER alongside its three variants across all projects. For detailed results, please refer to our open-source replication package [23]. The findings indicate that each key component or strategy substantially contributes to the effectiveness of TESTLINKER, as their exclusion results in lower F1-scores.

- **TESTLINKER-w/o C1**: This variant relies on our ranking model to construct TCTLs. Interestingly, it still maintains a precision of 73.31% and a recall of 58.21%, with an F1-score of 64.89%, which is notable given that test names are masked during training. Despite this, the model still extracts useful clues from the test names during inference. This phenomenon is likely attributable to the PCMs, which, during pre-training, might learn and capture semantic correlations between function names and their code functionalities. For example, in the `gson` project [71], the `testPrematureClose` examines the `Close` function. The model successfully identifies the focal method `Close` from the key token in the test name. But, with the test name masked, it incorrectly suggests `setLenient`.
- **TESTLINKER-w/o C2**: This variant theoretically yields the same performance to *Static NC* baseline. While **Component#1** is very precise, its limited scope results in a low recall. Given the high precision of the naming convention and the computational overhead required to invoke the model, the two-phase framework can enhance TESTLINKER's adaptability. Further discussion on **Component#1** is provided in Section 7.1.
- **TESTLINKER-w/o rules**: This variant achieves a precision of 67.84%, a recall of 57.31%, and an F1-score of 62.14%. The decline in performance underscores the importance of the filtering rules in improving prediction quality by excluding irrelevant function calls.

Answer to RQ2: Overall, when each of the components and strategies of the TESTLINKER is removed, the TCTL identification performance of the TESTLINKER is typically reduced to varying degrees, with **Component#2** and heuristic filtering rules significantly affecting the performance of the TESTLINKER.

6.3 RQ3: The Effect of Different Number of Recommended Function Names

6.3.1 Baselines

When resources are limited and there is restricted capacity for manually checking the recommended focal methods, it is more practical and efficient to focus on checking a smaller subset (*e.g.,* Top-1). Therefore, in **Component#2** of TESTLINKER, the model returns the function name with the highest prediction score. Ideally, each test should correlate directly with a single focal method. However, in practical scenarios, tests sometimes encompass multiple focal methods with different names. As described in Section 5.2.2, 19.91% of the tests in the manually labeled dataset involve testing multiple focal methods. To evaluate the impact of recommending different numbers of function names on TESTLINKER's performance, we build three variants of our approach: TESTLINKER-r2, TESTLINKER-r3, and TESTLINKER-r4. These variants are the same as TESTLINKER except for returning the top-2, top-3, and top-4 recommended function names, respectively. We then compare their performance with that of TESTLINKER in the manually labeled dataset.

6.3.2 Results

TABLE 5: Overall performance comparisons of different number of recommended functions

The number of recommended functions	Evaluation Metrics		
	Pre.	Recall	F1-score
TESTLINKER	**73.51%**	58.81%	**65.34%**
TESTLINKER-r2	58.55%	67.46%	62.69%
TESTLINKER-r3	50.52%	73.13%	59.76%
TESTLINKER-r4	45.24%	**75.22%**	56.50%

Table 5 presents the average results of various variants across all projects. We observe that TESTLINKER outperforms its variants (TESTLINKER-r2 to TESTLINKER-r4) in precision and F1-score, demonstrating its robust ability to recommend the most relevant focal method while minimizing false positives. As the variants recommend more function names, there is a noticeable increase in recall, albeit at the expense of precision. This compromise is reflected in the declining F1-scores with an increasing number of recommendations. Considering the trade-off between precision and recall, in this paper, we set the number of recommended focal method names to one.

Answer to RQ3: The experimental results demonstrate that TESTLINKER outperforms its variants (TESTLINKER-r2 to TESTLINKER-r4) in both precision and F1-score metrics, with fewer false positives. Furthermore, as the number of recommended function names increases, the recall of TESTLINKER exhibits a concurrent rise, albeit at the expense of reduced precision.

6.4 RQ4: The Effect of Code Model Choices

6.4.1 Baselines

To gain a deeper understanding of how different code models influence TESTLINKER's performance in identifying TCTLs, we evaluate four widely-used PCMs on the manually labeled dataset and *Testlink*, *i.e.,* CodeBERT [30], GraphCodeBERT [31], UniXcoder [44], and CodeT5 [16]. These baselines are selected based on the following criteria: 1) an extensive pre-training code corpus, 2) public availability, and 3) diverse model structures. Large language models (LLMs) [32], [72] trained on ultra-large-scale corpora

TABLE 6: The performance of PCMs in TCTL prediction

PCMs	Manually Labeled Dataset						*TestLink*		
	Positive	True Positive	False Negative	Pre.	Recall	F1-score	Pre.	Recall	F1-score
CodeBERT	269	195	140	72.49%	58.21%	64.57%	91.65%	91.87%	91.76%
GraphCodeBERT	267	193	142	72.28%	57.61%	64.12%	92.11%	91.52%	91.81%
UniXcoder	269	195	140	72.49%	58.21%	64.57%	93.17%	91.45%	92.30%
CHATGPT	264	189	146	71.59%	56.42%	63.11%	77.78%	50.06%	60.92%
CodeT5	268	197	138	**73.51%**	**58.81%**	**65.34%**	**93.55%**	**92.17%**	**92.86%**

have made significant strides and exhibit remarkable performance across various tasks. One noteworthy example is CHATGPT [73], developed by OpenAI, which stands out with its impressive 175 billion parameters and extensive training data. The substantial scale of CHATGPT empowers it with the ability to excel in understanding and generating text across diverse domains. Consequently, we further explore the potential of augmenting the TESTLINKER framework with CHATGPT's in-context learning capabilities to improve the TCTL identification.

Experimental Design. For each PCM, similar to CodeT5, we fine-tune it on the $TestLink$ training dataset to enable it to learn semantic associations between the test code and the focal method. Regarding CHATGPT [73], we access it through the gpt-3.5-turbo-0301 API, the latest available version. Interaction with CHATGPT involves natural language conversations, where requests are sent, and responses are received conversationally. To integrate CHATGPT into the TESTLINKER framework, we use tailored prompts that guide CHATGPT in generating responses that specify the most likely focal method name within the test. This process involves presenting the test to CHATGPT and highlighting the function call names embedded within it. In our experimental setup, each interaction with CHATGPT begins with the prompt: "*You are an experienced Java test engineer. In the following Java test method, please identify and output the corresponding focal method from the given ##Invocations, returning only the focal method's name without additional text.*" We replace the neural ranking model of TESTLINKER with the fine-tuned PCM and CHATGPT, respectively, to obtain TESTLINKER's performance in this configuration. In other words, PCMs and CHATGPT are used only to determine the function name of the most likely focal method for each test, while the rest of the process remains consistent with TESTLINKER.

6.4.2 Results

The results from Table 3 and Table 6 reveal that all TESTLINKER variants, based on different PCMs and CHATGPT, significantly outperform static TCTL baselines in F1-scores. For example, the UniXcoder-based variant still achieves an average F1-score enhancement ranging from 71.43% to 198.44%. Although these PCMs demonstrate similar performance in TCTL identification, the CodeT5-based variant slightly outperforms others. This finding is apparent on both the $TestLink$ and the manually labeled dataset. Specifically, CodeT5 exhibits improvements in precision, recall, and F1-score, albeit the increments are modest. On the $TestLink$ dataset, the increments range from 0.41% to 2.07% in precision, 0.33% to 0.79% in recall, and 0.60% to 1.20% in F1-score. Similarly, on the manually labeled dataset, the improvements are 1.40% to 1.69% in precision, 1.03% to 2.07% in the recall, and 1.19% to 1.90% in the F1-score. We attribute this optimal performance to CodeT5's extensive pre-training code corpus, which includes both CodeSearchNet and BigQuery datasets. In contrast, models like CodeBERT are solely dependent on the CodeSearchNet dataset. This evaluation demonstrates TESTLINKER's generalizability as a framework compatible with a variety of PCMs, not exclusively relying on CodeT5. Future work will focus on improving the semantic understanding of PCMs, thereby augmenting the effectiveness of TESTLINKER in TCTL identification.

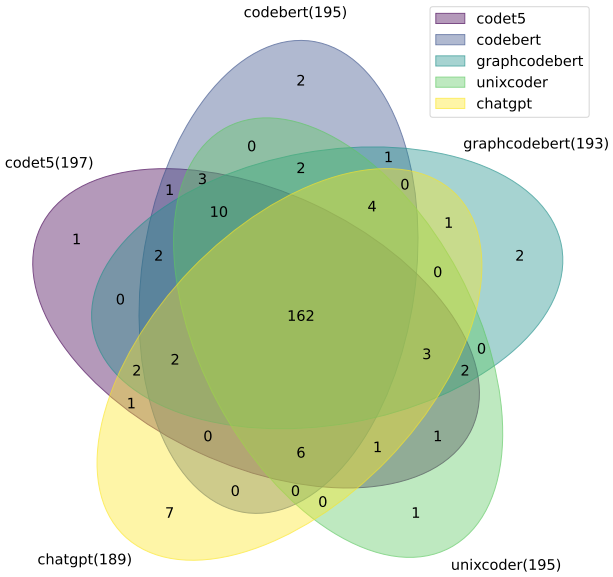

Fig. 4: The overlaps of the TCTL identified by different PCMs.

Table 6 presents the confusion matrix for CHATGPT and various PCMs in TCTL identification, where *Positive* indicates the total number of predicted focal methods, and *True Positive* refers to those correctly identified. *False Negative* represents the number of actual focal methods that the model fails to identify. Figure 4 illustrates the number of overlapping TCTLs identified by different PCMs and CHATGPT. From Table 6, it is evident that CHATGPT underperforms in identifying TCTLs compared to CodeT5, with an average precision of 71.59%, recall of 56.42%, and F1-score of 63.11% in the manually labeled dataset. Figure 4 shows that CHATGPT accurately identifies 189 TCTLs, slightly fewer than the 197 identified by CodeT5, 193 by GraphCodeBERT, and 195 by UniXcoder. These results highlight CHATGPT's limitations in general TCTL identification. However, CHATGPT uniquely identifies 7 TCTLs that other models miss, underscoring its distinct advantages in specific scenarios. This suggests that CHATGPT's unique strengths can complement traditional PCMs, potentially by

integrating its conversational understanding into a hybrid model or leveraging it in specific contexts to enhance TCTL identification.

The results in Table 6 show a notable discrepancy in the performance of different PCMs on the manually labeled dataset versus the $TestLink$ dataset. The manually labeled dataset presents more complex scenarios, as it includes 19.91% of tests covering multiple focal methods and 9.96% of tests lacking direct focal method calls. This complexity is reflective of real-world testing scenarios and poses significant challenges for TCTL identification methods. For the PCMs, the higher performance on the $TestLink$ dataset can be attributed to the dataset's design, which ensures that each test has a one-to-one correspondence with its focal method and that the focal methods appear directly in the test. This simplifies the task for the PCMs, allowing them to achieve higher precision and recall. However, CHATGPT exhibits an inverse performance trend, performing better on the manually labeled dataset than on the $TestLink$ dataset. This can be explained by CHATGPT's unique strength in understanding and generating text through conversational context, which allows it to handle more nuanced and complex scenarios better. In contrast, the $TestLink$ dataset's simpler and more controlled environment, designed specifically for training semantic associations, may not fully leverage CHATGPT's advanced conversational capabilities. The more rigid and straightforward structure of $TestLink$ aligns less with CHATGPT's strengths, thus leading to relatively poorer performance compared to other PCMs.

Answer to RQ4: 1) The experimental results indicate that all variants of TESTLINKER, utilizing different PCMs and CHATGPT, significantly outperform static TCTL baselines in F1-scores, with the CodeT5-based variant showing slightly superior performance. 2) The experimental results reveal that CHATGPT has lower precision, recall, and F1-score in TCTL identification, compared to CodeT5, GraphCodeBERT, and UniXcoder. However, CHATGPT uniquely identifies more TCTLs not detected by other models, suggesting its potential complementary role in specific scenarios or hybrid model integration.

7 DISCUSSION

7.1 Selection of Component#1

As shown in Table 4, when examining the performance comparison between **TESTLINKER-w/o C1** and TESTLINKER, the careful reader may notice that they exhibit similar levels of precision, recall, and F1-scores, with differences not exceeding 1%. This minimal variance suggests that replacing **Component#1** with another static-based method could potentially amplify TESTLINKER's effectiveness. However, our current framework aims to ensure identification precision in the initial phase, thereby placing the more challenging samples in the second phase. In the subsequent phase, the TESTLINKER employs semantic analysis for TCTL construction, yielding better identification performance. Despite other static methods achieving higher F1-scores, their precision significantly trails behind static NC. Therefore, we chose the static NC strategy for the initial phase to prioritize early-stage precision. This structured approach focuses on precision first, followed by advanced identification capabilities to handle complex samples, effectively managing varying complexities within the samples.

Test Code

```
@Test
public void testLanguagesByCountry() {
    assertLanguageByCountry(null, new String[0]);
    assertLanguageByCountry("GB", new String[]{"en"});
    assertLanguageByCountry("ZZ", new String[0]);
    assertLanguageByCountry("CH", new String[]{"fr", "de", "
    ↪ it"});
}
```

Fig. 5: A test sample that is successfully handled by **Component#1** in `Commons-Lang` project

Moreover, there are instances where the test code cannot be parsed to retrieve function calls, causing the ranking model to fail. However, some of these instances can be effectively handled using naming conventions. For example, as shown in Figure 5, the `testLanguagesByCountry` directly invokes a custom assertion function that calls the actual focal method `languagesByCountry()`. The neural ranking model might fail in this scenario because the function calls are not directly present in the test code but are nested within custom assertion methods. This nested call structure can obscure the direct link between the test and the focal method. However, naming conventions can succeed where the neural ranking model fails because they rely on the naming patterns of the tests and the methods. In this case, the test name `testLanguagesByCountry` closely matches the method name `languagesByCountry`, allowing the naming convention to establish the TCTL without parsing the nested function calls. In addition, static NC reduces the time cost and computational overhead for TESTLINKER. For the 335 manually labeled TCTLs, recognizing TCTLs using only **Component#2** takes 84.58 seconds. When combining **Component#2** with naming conventions, the time cost is reduced to 67.28 seconds, with improvements in both precision and recall. The naming convention effectively handles simple TCTL scenarios through rules, reducing the need for neural ranking model calls and enhancing the cost-effectiveness of TESTLINKER. Moving forward, we will explore further optimization of **Component#1** to enhance its capability and overall efficiency in recognizing TCTLs.

7.2 Application of Our Approach

TESTLINKER has the potential to facilitate automated software engineering research in various domains, such as obsolete test code detection (OTCD) [74], [27], regression testing (RT) [1], and bug localization (BL) [75], [76], *etc.*

Obsolete Test Code Detection (OTCD): TESTLINKER enhances OTCD by improving both precision and recall. Higher precision ensures superior data quality for model training, which enhances the accuracy of detection models. Increased recall reduces the likelihood of missing obsolete tests, ensuring that outdated or redundant test cases are efficiently identified and removed. This results in more maintainable and relevant test suites, streamlining the software maintenance process.

Regression Testing (RT): In the RT, TESTLINKER improves both precision and recall, optimizing the testing process. Enhanced precision directs developers' attention to the most relevant tests, reducing the overall testing cycle time. Additionally, increased recall expands test coverage, thereby improving the likelihood of detecting regressions. This contributes to higher software quality and reliability.

Bug Localization (BL): Bug localization involves identifying the exact location of bugs within the codebase after a test failure is detected. TESTLINKER can enhance bug localization efforts by accurately mapping test failures to the corresponding focal methods. This helps developers quickly pinpoint the root cause of a bug, thereby accelerating the debugging and fixing process. By providing precise links between tests and their focal methods, TESTLINKER reduces the time and effort required to diagnose issues, facilitating faster resolution of software defects.

These applications highlight the versatility and potential of TESTLINKER in improving various aspects of software testing and maintenance, ultimately contributing to higher software quality and more efficient development processes.

7.3 Limitation of Our Approach

We also investigate where TESTLINKER fails to correctly build TCTLs. We manually review the samples where TESTLINKER cannot make accurate predictions. The failures primarily stem from the following issues: 1) As discussed in RQ3, certain test cases lead to one-to-many scenarios, where the test code tests more than one focal method. By default, TESTLINKER returns only the Top-1 recommended function name, which may result in the omission of some positive samples for such test cases. 2) TESTLINKER struggles with detecting indirect method calls. However, in some instances, the test code does not directly call the focal method. As shown in Figure 6, in the `Commons IO` project, the `testTextXml` only calls the `checkTextXml` function [77], a private function within the test class which calls the focal method. At this stage, TESTLINKER, relying mainly on parsing direct function calls in test code, fails to recognize these indirect relationships. Addressing this issue would require TESTLINKER to trace indirect call chains, necessitating more advanced static analysis to deeply understand the call relationships in code. 3) Finally, we face challenges due to the limitations of `JavaParser`, which fails to parse some implicitly called functions, such as the `values` function of enumerated classes in `Commons IO` project [78]. Exploring solutions to the above problems presents an interesting and promising direction for future work.

7.4 Implication and Guideline

7.4.1 Implications for Practitioners

Considering Constructors as Focal Methods. In unit testing, constructors can serve as focal methods, especially when they are crucial for initializing the test environment. For example, our manually labeled dataset reveals that out of 335 TCTLs, 10 instances involve constructors as focal methods. This is particularly important in object-oriented programming, where constructors often play a significant role in setting up the initial state of an object and ensuring its proper configuration before any functions are invoked. Practitioners should recognize the importance of constructors and include them in their traceability analysis. By considering constructors as potential focal methods, practitioners can achieve a more comprehensive understanding of the test coverage and ensure that all critical initialization processes are adequately traced. This approach can lead to more thorough testing and better identification of issues related to object instantiation and state management.

Test Code

```
@Test
public void testTextXml() {
    checkTextXml(false, null);
    checkTextXml(false, "");
    checkTextXml(false, "text/atomxml");
}
```

Other Test Code

```
private void checkTextXml(final boolean expected,
                          final String mime) {
    assertEquals(expected, XmlStreamReader.isTextXml(mime),
                 "Mime=[" + mime + "]");
}
```

Fig. 6: A failure of TESTLINKER in `Commons-IO` project

Handling Third-Party Dependencies. In real-world projects, tests often invoke methods from third-party dependencies. While our manually labeled dataset does not reveal cases where third-party library functions are the focal methods for tests, TESTLINKER does consider third-party APIs within its neural ranking model. Specifically, we exclude only Java standard and JUnit APIs from our analysis. This exclusion is a deliberate choice aimed at reducing noise and improving the precision of the traceability links. Java standard APIs often include utility functions that are widely used across various parts of the codebase, which could lead to an overwhelming number of false positives if included. Similarly, assertion APIs are used to verify conditions in tests but do not themselves represent the core functionality being tested. By filtering out these common and generic APIs, TESTLINKER focuses on identifying more meaningful traceability links that are directly relevant to the application's unique logic and functionality.

Ensuring that all third-party APIs used in tests are included in the traceability analysis is essential for a complete and accurate mapping of test cases. This comprehensive approach helps in identifying potential issues arising from third-party interactions, thereby improving the overall reliability and maintainability of the software. Practitioners should extend their traceability analysis to include third-party dependencies, ensuring a complete and accurate mapping of test cases to all relevant code. This includes identifying direct calls to third-party methods and understanding the interactions and dependencies that these methods introduce. By incorporating third-party libraries into the traceability analysis, practitioners can uncover hidden dependencies and ensure that all parts of the software ecosystem are adequately tested and traced.

7.4.2 Implications for Researchers

Addressing Mocks and Stubs. Following previous research [3], [15], our current training and evaluation datasets are designed for the Java programming language and utilize

Test Code

```
/**
 * Test localeLookupList() method.
 */
@Test
public void testLocaleLookupList_Locale() {
    assertLocaleLookupList(null, null, new Locale[0]);
    assertLocaleLookupList(LOCALE_QQ, null, new Locale[{
            LOCALE_QQ});
    assertLocaleLookupList(LOCALE_EN, null, new Locale[]{
            LOCALE_EN});
    assertLocaleLookupList(LOCALE_EN, null, new Locale[]{
            LOCALE_EN});
    assertLocaleLookupList(LOCALE_EN_US, null,
        new Locale[] {
            LOCALE_EN_US,
            LOCALE_EN});
    assertLocaleLookupList(LOCALE_EN_US_ZZZZ, null,
        new Locale[] {
            LOCALE_EN_US_ZZZZ,
            LOCALE_EN_US,
            LOCALE_EN});
}
```

Fig. 7: A test sample in `Commons-Lang` project

the JUnit testing framework. While these datasets have been meticulously curated, they do not explicitly account for the presence of mocks and stubs within the tests. Mocks and stubs are commonly used in unit testing to simulate dependencies and isolate the functionality of the unit under test. Their presence in testing scenarios can complicate the identification of focal methods in TCTL, as the approach must distinguish between real method calls and those handled by mocks or stubs. To address this limitation, researchers could focus on explicitly identifying and handling mocks and stubs within these testing scenarios. This enhancement would involve: 1) **Detection and Annotation**: Developing techniques to automatically detect and annotate mocks and stubs in the test code. This could involve static analysis to identify common mocking frameworks like `Mockito` [79], `EasyMock` [80], and `JMock` [81]. 2) **Impact Analysis**: Investigating the impact of mocks and stubs on the accuracy of TCTL identification. This would involve comparing the performance of TCTL methods on tests with and without mocks and stubs. 3) **Refinement of TCTL Methods**: Adapting TCTL methods to handle mocks and stubs appropriately. This could involve enhancing the algorithms to recognize when a method call is to a mock or stub and adjusting the traceability links accordingly.

Enhancing TCTL Identification with Additional Context. Comments and Javadoc often provide valuable insights that can significantly improve the accuracy of constructing traceability links. For example, as shown in Figure 7, in the `Commons Lang` project, a test with a comment explicitly stating that it tests the `localeLookupList()` method provides a clear indication of the focal method. This observation suggests that leveraging such contextual information can greatly enhance TCTL identification. Researchers could consider developing heuristic rules to parse comments and Javadoc for keywords and phrases that indicate the focal method. Patterns such as `tests`, `checks`, or `verifies` followed by a method name can be valuable indicators. Additionally, integrating this context into neural ranking models can further enhance the ability to identify TCTLs by recognizing and leveraging the semantic information provided by comments and Javadoc. By incorporating this additional context, tools like TESTLINKER can better handle cases where the test does not follow naming conventions or lacks direct function calls to the focal method. This approach can lead to more accurate and reliable traceability link construction, ultimately improving various aspects of software testing and maintenance.

Enhancing TCTL Identification for Multi-Focal Method Recognition. Currently, TESTLINKER recommends only the most likely focal method name to maintain high precision in TCTL identification. While this approach effectively reduces false positives, it may lead to the omission of valid TCTLs, especially in real-world scenarios where a test might involve multiple focal methods with different names. This observation highlights an important consideration for researchers: the trade-off between precision and recall in TCTL identification. An intuitive approach might suggest that the number of assertions in a test correlates positively with the number of focal methods. However, our analysis shows that this is not strongly supported. Specifically, in tests with multiple focal methods, the number of assertions does not strongly correlate with the number of focal methods, as indicated by an R^2 value (determination coefficient) of 0.2282. This suggests that relying solely on the number of assertions to predict multiple focal methods may not be accurate. Therefore, researchers need to explore more effective methods, such as adaptive thresholding techniques, to dynamically adjust the number of recommended focal methods based on the confidence scores and the complexity of the test code.

8 THREATS TO VALIDITY

Internal Validity. Threats to internal validity refer to the biases in our experiments. In our work, we compare TESTLINKER with TCTL methods across different categories. The implementation and execution of these baselines could potentially threaten internal validity. To mitigate this threat, we directly re-ran the TCTL tools using their default parameter configurations. Additionally, we carefully checked the source code of TESTLINKER and the baselines, and we publicly released all materials for further validation. In RQ1-RQ3, we manually construct an additional evaluation dataset. The use of manual analysis could suffer from subjectivity bias when interpreting which functions are tested by a test. To mitigate this threat, we employ three volunteers to conduct a two-phase verification process. Although the volunteers share the same student status, they come from diverse backgrounds and possess extensive prior experience. Each volunteer independently assesses the tests, ensuring a variety of perspectives. In cases of disagreement, a conference is held to discuss and resolve differences. The number of disagreements is small, and the smallest changes enacted during the conference are to correct erroneous results rather than to convince volunteers to change their judgments. This approach ensures that our evaluation is robust and minimizes the risk of subjectivity bias, thereby enhancing the reliability of our findings.

External Validity. Threats to external validity involve the generalization of TESTLINKER. Our dataset, derived from Java projects, might raise concerns about TESTLINKER's applicability to other programming languages. However, Java is one of the most popular programming languages.

Furthermore, the underlying ranking model, based on the PCMs pre-trained with the code from various programming languages, is adaptable to diverse programming contexts. While TESTLINKER incorporates Java-specific heuristic rules, these rules can be easily modified for other languages. For instance, mainstream Python testing frameworks like pytest [82] also follow naming conventions. Python, similar to Java, possesses built-in functions and inheritance structures useful for pruning function call lists during model inference. Our evaluation datasets, recognized for considerable size, demonstrate TESTLINKER's suitability for complex software systems. Additionally, these datasets have been scrupulously annotated by three judges to minimize manual evaluation biases [3], [15]. Finally, we have expanded our evaluation dataset to include a diverse range of projects beyond utility libraries. The inclusion of Jenkins (an automation server) and Dubbo (an RPC framework) ensures that our dataset covers various project types and application scenarios. However, despite these efforts, the dataset may still not capture the full spectrum of possible software projects. Future work could involve extending the dataset further to include additional domains such as mobile applications, web applications, and embedded systems to enhance the generalizability of our findings.

9 Related Work

Traceability link recovery (TLR) is a fundamental aspect of software engineering, enabling the establishment of connections between various artifacts generated during the software development lifecycle. While our work with TESTLINKER focuses on constructing traceability links from tests to code, it is essential to consider other significant approaches in this domain and compare them to our approach.

Marcus and Maletic [83] use latent semantic indexing (LSI) to recover traceability links between software artifacts. While effective, LSI-based approaches can suffer from precision and recall limitations due to the ambiguity and variability in natural language descriptions. TESTLINKER improves on this by using a combination of static analysis and a neural ranking model to capture both syntactic and semantic information, enhancing the accuracy of traceability link construction. Murphy *et al.* [84] explore the recovery of transitive traceability links among software artifacts. Their approach, called the Connecting Links Method (CLM), extends traditional traceability by identifying indirect relationships between artifacts and using a third artifact to bridge the connection. For example, CLM can link requirement $R1$ to source code $S1$ via design $D1$, even if a direct link between $R1$ and $S1$ is weak or missing. This method improves the coverage of traceability links where direct connections are insufficient. Furtado *et al.* [85] propose a technique, namely Trace++, to assist with the transition from traditional to agile methodologies by extending traditional traceability relationships with additional information. This work focuses on addressing specific problems, such as the lack of metrics for rework, understanding high-level project scope, documenting non-functional requirements, and maintaining management control during the transition to agile processes. While Trace++ provides support for transitioning between methodologies and handling broader development artifacts, TESTLINKER focuses on enhancing test-to-code traceability in both agile and traditional development environments. Rath *et al.* [86] recently addressed the problem of missing links between commits and issues in version control systems. Their approach uses a combination of process and text-related features to train a classifier that identifies missing issue tags in commit messages, thereby generating the missing links. This work focuses on enhancing the traceability between feature requests, bug fixes, commits, source code, and specific developers. This distinction highlights TESTLINKER's targeted approach to linking tests directly to the focal methods. Moran *et al.* [87] leverage Hierarchical Bayesian Networks (HBNs) to enhance the recovery of traceability links between software artifacts such as requirements, design documents, and code. The hierarchical nature of HBNs allows for the integration of multiple sources of evidence and their dependencies, resulting in more accurate and reliable traceability links. In contrast, TESTLINKER specifically targets test-to-code traceability, thus focusing on a different aspect of the traceability problem.

In summary, while numerous methods have been proposed for TLR, ranging from information retrieval techniques to machine learning and hierarchical Bayesian networks, TESTLINKER offers a novel approach by combining static analysis with neural ranking and heuristic refinement. This comprehensive strategy allows TESTLINKER to effectively handle the specific challenges of test-to-code traceability, setting it apart from other methods in the field.

10 Conclusion and Future Work

In this paper, we propose TESTLINKER, a novel approach designed to enhance the establishment of test-to-code traceability links (TCTLs). To the best of our knowledge, we are the first to use a two-phase framework to identify TCTLs, combining heuristic rules and semantic learning capabilities of Pre-trained Code Models (PCMs) to address this complex challenge. We build TESTLINKER using the pre-trained CodeT5 model and fine-tune CodeT5 for TCTL. For training TESTLINKER, we construct an adapted dataset. The evaluation results demonstrate that TESTLINKER significantly outperforms all static baselines. Moreover, when compared to the state-of-the-art dynamic TCTL methods, TESTLINKER achieves comparable or even better performance in identifying traceability links. Future work can explore enhancing our approach by integrating contextual information and advancing learning techniques for code semantics. Additionally, we aim to extend our approach to other programming languages, such as Python and JavaScript, to broaden its applicability and generalizability.

11 Acknowledgements

We appreciate the insightful insights provided by anonymous reviewers to improve the quality of the paper. This work was supported in part by the National Natural Science Foundation of China (No. 62372071), the Chongqing Technology Innovation and Application Development Project (No. CSTB2022TIAD-STX0007 and No. CSTB2023TIAD-STX0025), the Natural Science Foundation of Chongqing (No. CSTB2023NSCQ-MSX0914) and the Fundamental Research Funds for the Central Universities (No. 2023CDJKYJH013).

REFERENCES

[1] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley, "Recovering test-to-code traceability using slicing and textual analysis," *Journal of Systems and Software*, vol. 88, pp. 147–168, 2014.

[2] R. Watkins and M. Neal, "Why and how of requirements tracing," *Ieee Software*, vol. 11, no. 4, pp. 104–106, 1994.

[3] R. White, J. Krinke, and R. Tan, "Establishing multilevel test-to-code traceability links," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 861–872.

[4] A. Qusef, G. Bavota, R. Oliveto, A. D. Lucia, and D. Binkley, "Evaluating test-to-code traceability recovery methods through controlled experiments," *Journal of Software: Evolution and Process*, vol. 25, no. 11, pp. 1167–1191, 2013.

[5] V. Csuvik, A. Kicsi, and L. Vidács, "Evaluation of textual similarity techniques in code level traceability," in *Computational Science and Its Applications–ICCSA 2019: 19th International Conference, Saint Petersburg, Russia, July 1–4, 2019, Proceedings, Part IV 19*. Springer, 2019, pp. 529–543.

[6] ——, "Source code level word embeddings in aiding semantic test-to-code traceability," in *2019 IEEE/ACM 10th International Symposium on Software and Systems Traceability (SST)*. IEEE, 2019, pp. 29–36.

[7] M. Gethers, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "On integrating orthogonal information retrieval methods to improve traceability recovery," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 133–142.

[8] A. Kicsi, L. Tóth, and L. Vidács, "Exploring the benefits of utilizing conceptual information in test-to-code traceability," in *Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, 2018, pp. 8–14.

[9] B. Van Rompaey and S. Demeyer, "Establishing traceability links between unit test cases and units under test," in *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 2009, pp. 209–218.

[10] P. Bouillon, J. Krinke, N. Meyer, and F. Steimann, "Ezunit: A framework for associating failed unit tests with potential programming errors," in *Agile Processes in Software Engineering and Extreme Programming: 8th International Conference, XP 2007, Como, Italy, June 18-22, 2007. Proceedings 8*. Springer, 2007, pp. 101–104.

[11] M. Ghafari, C. Ghezzi, and K. Rubinov, "Automatically identifying focal methods under test in unit test cases," in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2015, pp. 61–70.

[12] V. Hurdugaci and A. Zaidman, "Aiding software developers to maintain developer tests," in *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 2012, pp. 11–20.

[13] H. M. Sneed, "Reverse engineering of test cases for selective regression testing," in *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.* IEEE, 2004, pp. 69–74.

[14] A. Qusef, R. Oliveto, and A. De Lucia, "Recovering traceability links between unit tests and classes under test: An improved method," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.

[15] R. White and J. Krinke, "Tctracer: Establishing test-to-code traceability links using dynamic and static techniques," *Empirical Software Engineering*, vol. 27, no. 3, p. 67, 2022.

[16] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[17] A. Zaidman, B. Van Rompaey, A. Van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Software Engineering*, vol. 16, pp. 325–364, 2011.

[18] Z. Liu, K. Liu, X. Xia, and X. Yang, "Towards more realistic evaluation for neural test oracle generation," *arXiv preprint arXiv:2305.17047*, 2023.

[19] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers and focal context," vol. abs/2009.05617, 2020. [Online]. Available: https://arxiv.org/abs/2009.05617

[20] N. Rao, K. Jain, U. Alon, C. L. Goues, and V. J. Hellendoorn, "Cat-lm: Training language models on aligned code and tests," *arXiv preprint arXiv:2310.01602*, 2023.

[21] F. Hassan and X. Wang, "Hirebuild: An automatic approach to history-driven repair of build scripts," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 1078–1089.

[22] Y. Lou, Z. Chen, Y. Cao, D. Hao, and L. Zhang, "Understanding build issue resolution in practice: symptoms and fix patterns," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 617–628.

[23] 2024. [Online]. Available: https://figshare.com/s/6d9a729c2ebb83c4b291

[24] J. H. Hayes, A. Dekhtyar, and D. S. Janzen, "Towards traceable test-driven development," in *ICSE Workshop on Traceability in Emerging Forms of Software Engineering, TEFSE@ICSE 2009. Vancouver, BC, Canada, 18 May, 2009*. IEEE Computer Society, 2009, pp. 26–30.

[25] G. Rothermel and M. J. Harrold, "Empirical studies of a safe regression test selection technique," *IEEE Transactions on Software Engineering*, vol. 24, no. 6, pp. 401–419, 1998.

[26] Y. Lou, J. Chen, L. Zhang, and D. Hao, "A survey on regression test-case prioritization," in *Advances in Computers*. Elsevier, 2019, vol. 113, pp. 1–46.

[27] S. Wang, M. Wen, Y. Liu, Y. Wang, and R. Wu, "Understanding and facilitating the co-evolution of production and test code," in *2021 IEEE International conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2021, pp. 272–283.

[28] W. Sun, M. Yan, Z. Liu, X. Xia, Y. Lei, and D. Lo, "Revisiting the identification of the co-evolution of production and test code," *ACM Trans. Softw. Eng. Methodol.*, jul 2023, just Accepted.

[29] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, 2017, pp. 5998–6008.

[30] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, ser. Findings of ACL, vol. EMNLP 2020. Association for Computational Linguistics, 2020, pp. 1536–1547.

[31] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*, 2021.

[32] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[33] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*, 2023.

[34] W. Yuan, Q. Zhang, T. He, C. Fang, N. Q. V. Hung, X. Hao, and H. Yin, "CIRCLE: continual repair across programming languages," in *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, S. Ryu and Y. Smaragdakis, Eds. ACM, 2022, pp. 678–690.

[35] Y. Wu, N. Jiang, H. V. Pham, T. Lutellier, J. Davis, L. Tan, P. Babkin, and S. Shah, "How effective are neural networks for fixing security vulnerabilities," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, R. Just and G. Fraser, Eds. ACM, 2023, pp. 1282–1294.

[36] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Q. Phung, "Vulrepair: a t5-based automated software vulnerability repair," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, A. Roychoudhury, C. Cadar, and M. Kim, Eds. ACM, 2022, pp. 935–947.

[37] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *19th IEEE/ACM International*

Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022. ACM, 2022, pp. 608–620.

[38] B. Steenhoek, M. M. Rahman, R. Jiles, and W. Le, "An empirical study of deep learning models for vulnerability detection," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2237–2248.

[39] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk, and G. Bavota, "Using pre-trained models to boost code review automation," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 2291–2302.

[40] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu, and N. Sundaresan, "Automating code review activities by large-scale pre-training," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, A. Roychoudhury, C. Cadar, and M. Kim, Eds. ACM, 2022, pp. 1035–1047.

[41] M. Tufano, D. Drain, A. Svyatkovskiy, and N. Sundaresan, "Generating accurate assert statements for unit test cases using pretrained transformers," in *IEEE/ACM International Conference on Automation of Software Test, AST@ICSE 2022, Pittsburgh, PA, USA, May 21-22, 2022*. ACM/IEEE, 2022, pp. 54–64.

[42] A. Mastropaolo, S. Scalabrino, N. Cooper, D. Nader-Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Studying the usage of text-to-text transfer transformer to support code-related tasks," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 336–347.

[43] A. Mastropaolo, N. Cooper, D. Nader-Palacio, S. Scalabrino, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Using transfer learning for code-related tasks," *IEEE Trans. Software Eng.*, vol. 49, no. 4, pp. 1580–1598, 2023.

[44] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Association for Computational Linguistics, 2022, pp. 7212–7225.

[45] "`Commons-math` project." https://github.com/apache/commons-math/blob/3.6-release/src/test/java/org/apache/commons/math3/linear/DiagonalMatrixTest.java, 2016.

[46] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[47] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction," *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1267–1293, 2018.

[48] L. Zhang, X. Zhang, and J. Pan, "Hierarchical cross-modality semantic correlation learning model for multimodal summarization," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 10, 2022, pp. 11 676–11 684.

[49] Q. Zhang, C. Fang, W. Sun, Y. Liu, T. He, X. Hao, and Z. Chen, "Appt: Boosting automated patch correctness prediction via fine-tuning pre-trained models," *IEEE Transactions on Software Engineering*, 2024.

[50] "`StringTest` in gson," https://github.com/google/gson/blob/main/gson/src/test/java/com/google/gson/functional/StringTest.java, 2023.

[51] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," *arXiv preprint arXiv:1508.07909*, 2015.

[52] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, 2019, pp. 4171–4186.

[53] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[54] T. J. Team, "JavaParser: A java parser library," 2023, software available from https://javaparser.org.

[55] "An example in commons-io project." https://github.com/apache/commons-io/blob/master/src/test/java/org/apache/commons/io/FileUtilsTest.java, 2024.

[56] M. Tufano, S. K. Deng, N. Sundaresan, and A. Svyatkovskiy, "METHODS2TEST: A dataset of focal methods mapped to test cases," in *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. ACM, 2022, pp. 299–303.

[57] V. Lenarduzzi, N. Saarimäki, and D. Taibi, "Some sonarqube issues have a significant but small effect on faults and changes. a large-scale empirical study," *Journal of Systems and Software*, vol. 170, p. 110750, 2020.

[58] J. Tan, D. Feitosa, and P. Avgeriou, "Does it matter who pays back technical debt? an empirical study of self-fixed td," *Information and Software Technology*, vol. 143, p. 106738, 2022.

[59] "About stars (GitHub)." https://docs.github.com/en/get-started/exploring-projects-on-github/saving-repositories-with-stars, 2021.

[60] F. Wen, C. Nagy, G. Bavota, and M. Lanza, "A large-scale empirical study on code-comment inconsistencies," in *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 2019, pp. 53–64. [Online]. Available: https://doi.org/10.1109/ICPC.2019.00019

[61] "Project jenkins." https://github.com/jenkinsci/jenkins, 2024.

[62] "Project dubbo." https://github.com/apache/dubbo, 2024.

[63] "PyTorch," https://pytorch.org/., 2023.

[64] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[65] M. Madeja and J. Porubän, "Tracing naming semantics in unit tests of popular github android projects," in *8th Symposium on Languages, Applications and Technologies, SLATE 2019, June 27-28, 2019, Coimbra, Portugal*, ser. OASIcs, vol. 74, 2019, pp. 3:1–3:13.

[66] H. Schütze, C. D. Manning, and P. Raghavan, *Introduction to information retrieval*. Cambridge University Press Cambridge, 2008, vol. 39.

[67] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th international conference on Software engineering*, 2002, pp. 467–477.

[68] R. M. Parizi, S. P. Lee, and M. Dabbagh, "Achievements and challenges in state-of-the-art software traceability between test and code artifacts," *IEEE Transactions on Reliability*, vol. 63, no. 4, pp. 913–926, 2014.

[69] F. Wilcoxon, *Individual comparisons by ranking methods*. Springer, 1992.

[70] N. Cliff, *Ordinal methods for behavioral data analysis*. Psychology Press, 2014.

[71] "`JsonTreeWriterTest` in gson," https://github.com/google/gson/blob/gson-parent-2.8.0/gson/src/test/java/com/google/gson/internal/bind/JsonTreeWriterTest.java, 2023.

[72] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. T. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, "OPT: open pre-trained transformer language models," *CoRR*, vol. abs/2205.01068, 2022.

[73] J. Schulman, B. Zoph, C. Kim, J. Hilton, J. Menick, J. Weng, J. Uribe, L. Fedus, L. Metz, M. Pokorny *et al.*, "Chatgpt: Optimizing language models for dialogue," 2022.

[74] X. Hu, Z. Liu, X. Xia, Z. Liu, T. Xu, and X. Yang, "Identify and update test cases when production code changes: A transformer-based approach," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1111–1122.

[75] M. Pradel, V. Murali, R. Qian, M. Machalica, E. Meijer, and S. Chandra, "Scaffle: Bug localization on millions of files," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 225–236.

[76] B. Wang, L. Xu, M. Yan, C. Liu, and L. Liu, "Multi-dimension convolutional neural network for bug localization," *IEEE Transactions on Services Computing*, vol. 15, no. 3, pp. 1649–1663, 2020.

[77] "`Commons-IO` project, `testTextXml` function." https://github.com/apache/commons-io/blob/master/src/test/java/org/apache/commons/io/input/XmlStreamReaderUtilitiesTest.java, 2023.

[78] "`Commons-IO` project, `test_serialization` function." https://github.com/apache/commons-io/blob/commons-io-2.5/src/test/java/org/apache/commons/io/IOCaseTestCase.java, 2023.

[79] Mockito Contributors, "Mockito: A framework for unit tests in java," https://site.mockito.org/, 2024, https://site.mockito.org/.
[80] EasyMock Team, "Easymock: A library that provides an easy way to use mock objects for unit testing," https://easymock.org/, 2024, https://easymock.org/.
[81] JMock Project, "Jmock: A library for testing java code with mock objects," http://jmock.org/, 2024, http://jmock.org/.
[82] "Pytest." https://pytest.org/en/latest/explanation/goodpractices.html#test-discovery, 2021.
[83] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 125–135.
[84] K. Nishikawa, H. Washizaki, Y. Fukazawa, K. Oshima, and R. Mibe, "Recovering transitive traceability links among software artifacts," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 576–580.
[85] F. Furtado and A. Zisman, "Trace++: A traceability approach to support transitioning to agile software engineering," in *2016 IEEE 24th International Requirements Engineering Conference (RE)*. IEEE, 2016, pp. 66–75.
[86] M. Rath, J. Rendall, J. L. Guo, J. Cleland-Huang, and P. Mäder, "Traceability in the wild: automatically augmenting incomplete trace links," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 834–845.
[87] K. Moran, D. N. Palacio, C. Bernal-Cárdenas, D. McCrystal, D. Poshyvanyk, C. Shenefiel, and J. Johnson, "Improving the effectiveness of traceability link recovery using hierarchical bayesian networks," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 873–885.