Reflective Unit Test Generation for Precise Type Error Detection with Large Language Models

Chen Yang[§], Ziqi Wang[§], Yanjie Jiang[§], Lin Yang[§], Yuteng Zheng[§], Jianyi Zhou[†], Junjie Chen[§*]
[§]College of Intelligence and Computing, Tianjin University, Tianjin, China
[†]Huawei Cloud Computing Technologies Co., Ltd., Beijing, China
{yangchenyc, wangziqi123, yanjiejiang, linyang, zyt_767904, junjiechen}@tju.edu.cn, zhoujianyi2@huawei.com

***Abstract*—Type errors in Python often lead to runtime failures, posing significant challenges to software reliability and developer productivity. Existing static analysis tools aim to detect such errors without execution but frequently suffer from high false positive rates. Recently, unit test generation techniques offer great promise in achieving high test coverage, but they often struggle to produce bug-revealing tests without tailored guidance. To address these limitations, we present RTED, a novel type-aware test generation technique for automatically detecting Python type errors. Specifically, RTED combines step-by-step type constraint analysis with reflective validation to guide the test generation process and effectively suppress false positives. We evaluated RTED on two widely-used benchmarks, BugsInPy and TypeBugs. Experimental results show that RTED can detect 22∼29 more benchmarked type errors than four state-of-the-art techniques. RTED is also capable of producing fewer false positives, achieving an improvement of 173.9%∼245.9% in precision. Furthermore, we applied RTED to six real-world open-source Python projects, and successfully discovered 12 previously unknown type errors, demonstrating RTED's practical value.**

***Index Terms*—Test Generation, Type Error, Bug Detection**

I. Introduction

Type errors are among the most common and critical issues in Python applications. They can lead to unexpected crashes and pose significant risks to the reliability of systems across various domains, including artificial intelligence platforms, data science pipelines, and financial applications. Despite their severity, type errors remain highly prevalent. According to Oh et al. [1], they account for over 30% of Python-related questions on Stack Overflow and GitHub issues, underscoring the urgent need for effective techniques to detect such errors.

To address this, several static techniques have been proposed for detecting type errors [1], [2]. However, these techniques often suffer from high false positive rates, which significantly limit their practicality. For example, PyInder [1], one of the most advanced static type checkers, reported tens of thousands of warnings but identified only 34 real type errors. This overwhelming noise renders such tools impractical for developers, who cannot afford to spend excessive time triaging false alarms. In contrast, unit testing is generally considered to produce fewer false positives in software quality assurance [3]. However, existing unit test generation techniques (including search-based [4] and LLM-based techniques [5]–[7]) typically use code coverage as their testing guidance, which is not directly aligned with the goal of bug detection and may thus limit their effectiveness in uncovering bugs (particularly for certain types of bugs such as type errors). Indeed, prior studies have shown that high coverage does not necessarily translate into effective bug detection [4], [8], [9]. Also, as demonstrated in our empirical study (presented in Section V), the state-of-the-art unit test generation technique (i.e., HITS [7]) was only able to detect 12 type errors out of the 69 benchmarked bugs.

Intuitively, shifting the testing guidance from enhancing code coverage to directly targeting bug detection could be helpful. Particularly, this shift is more feasible for LLM-based unit test generation, which can be guided through lightweight prompting, compared to search-based techniques that typically require significant engineering effort. Moreover, search-based techniques could struggle with generalizability across Python's diverse versions and its evolving type system. Therefore, our work focuses on leveraging LLM-based unit test generation to enhance the detection of type errors. However, simply instructing LLMs to generate unit tests for detecting type errors does not yield satisfactory effectiveness.

On one hand, LLMs are primarily trained on non-buggy code paired with passing tests, which leads them to favor safe and common usage patterns and test inputs, thereby limiting their ability to uncover type errors. While explicitly prompting the LLM to generate bug-revealing inputs may seem like a viable solution, on the other hand, it suffers from another challenge of context-aware type constraints in Python. Specifically, LLMs often lack awareness of such constraints, which typically arise from broader usage contexts that are not evident when analyzing the focal method in isolation. Violating these constraints leads to invalid crashes that do not reveal type errors, but instead indicate improper or unsupported method invocations. For example, as shown in Fig 1 (Test 2), the LLM generates a unit test for the method `_validate_key`. This test passes a `NumPy` array containing a `NaN` value to the method, resulting in a crash. However, in actual usage, the input is first validated by the method `has_valid_item`, which ensures that each element supports certain required magic methods, such as `__bool__`. Since `NaN` lacks these methods, it violates the context-aware constraint that the LLM fails to capture when focusing solely on the focal method. This leads to an invalid crash — effectively a false positive in type error detection. Moreover, LLM hallucinations can further exacerbate the issue of false positives.

* corresponding author

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00233

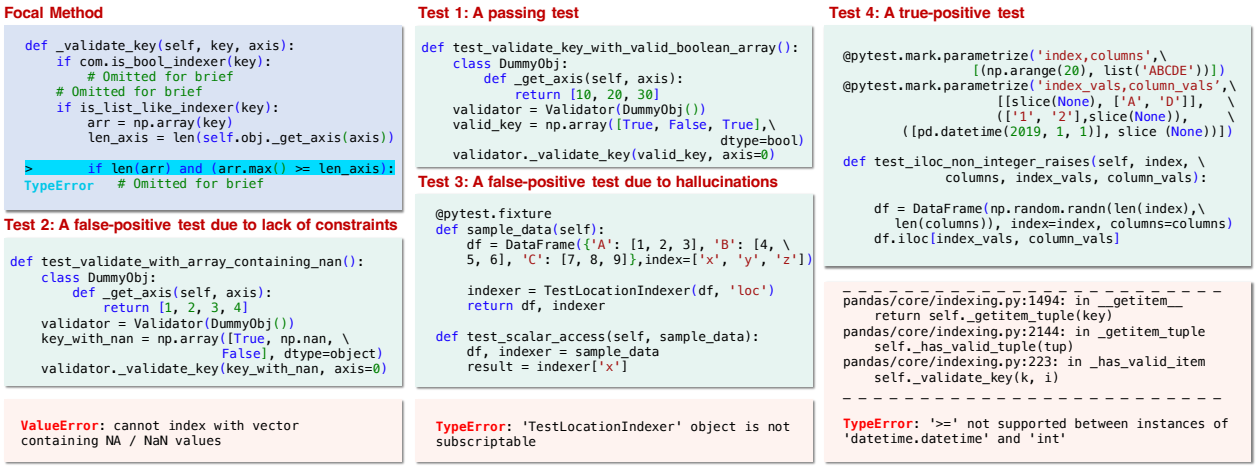

Fig. 1: Motivating example

To achieve precise type error detection, we propose **RTED** (**R**eflective **T**ype **E**rror **D**etection with LLMs), a novel technique that guides LLM-based unit test generation with type constraint analysis and self-reflection. First, RTED captures context-aware type constraints by analyzing the invocation chains leading to the focal method, enabling step-by-step backward propagation of these constraints. To enhance its effectiveness in uncovering type errors, RTED incorporates an error-seeking agent that performs type constraint propagation analysis: it identifies input types likely to trigger type errors in the focal method and infers the corresponding constraints that earlier methods in the invocation chain must satisfy to produce these inputs. By taking the inferred constraints, the invocation chain, along with the context information of the entry method as prompt, RTED then guides the LLM to generate bug-revealing tests for the invocation chain.

Due to the hallucination issues inherent in LLMs, the triggered failures may correspond to either true bugs or false positives. To mitigate the impact of false positives, RTED incorporates a self-reflection mechanism comprising three specialized agents: a type-consistency-checking agent, a semantic-validity-checking agent, and a meta-evaluation agent. The type-consistency-checking agent verifies whether the input types of the generated test conform to the constraints derived from type constraint analysis. The semantic-validity-checking agent assesses whether the generated test adheres to semantic expectations, such as contextual constraints or usage patterns that would not occur in realistic scenarios. The meta-evaluation agent integrates the judgments from the other two agents to determine whether a failure represents a genuine type error or a false positive. If a false positive is inferred, RTED uses this reflective feedback to refine its test generation process, enabling iterative improvement and enhancing the precision of the generated tests.

We evaluated RTED on two widely-used benchmarks in this field, i.e., BugsInPy [10] and TypeBugs [11], involving 16 Python projects with 69 real-world type errors. We compared RTED with the state-of-the-art Python type error detection technique (i.e., Pyinder [1]) and the state-of-the-art LLM-based unit test generation techniques (i.e., CHATTESTER [5], SymPrompt [6], and HITS [7]). Specifically, RTED detects 34 type errors with 13 false positives, while Pyinder, CHATTESTER, SymPrompt, and HITS detect only 5, 7, 3, and 12 bugs, with 21, 19, 14, 35 false positives, respectively. That is , RTED is not only capable of detecting 22∼29 more type errors than compared techniques, but also achieves an improvement of 173.9%∼245.9% in precision. Furthermore, we applied RTED to the latest versions of six popular open-source Python projects, and discovered 12 previously unknown type errors. These results demonstrate the effectiveness of RTED in uncovering type-related bugs through generating effective unit tests.

In summary, our contributions are as follows:

- We design RTED, a novel technique that leverages context-aware type constraint analysis combined with a self-reflection mechanism to guide LLM-based unit test generation, enabling precise Python type error detection.
- RTED introduces a type constraint analysis method that captures context-aware constraints via invocation chain analysis and employs an error-seeking agent to propagate constraints backward step-by-step, guiding the generation of targeted test inputs more likely to expose type errors.
- RTED incorporates a self-reflection process consisting of three specialized agents — type consistency checking, semantic validity checking, and meta-evaluation — to validate generated failures and iteratively refine test generation, reducing false positives in type error detection.
- We evaluated RTED on real-world Python projects, significantly outperforming state-of-the-art baselines on benchmarked bugs and detecting 12 previously unknown type errors in the wild.

II. Motivation

We illustrate the challenges of detecting Python type errors with a motivating example and describe how our technique mitigates them. The motivating example is presented in Figure 1, sampled from the real-world open-source project Pandas [12]. In this example, the type error is triggered by the operation `arr.max() ≥ len_axis`, where incompatible operand types can lead to a runtime crash (`len_axis` is an `int`, but `arr.max()` may return a non-integer type depending on the `key` argument).

To detect this bug, we need a test that exercises the faulty code. First, we used CHATTESTER's default prompt to instruct the state-of-the-art LLM, DeepSeek-V3 [13], to generate a test, which we refer to as `Test 1`. In `Test 1`, the input is a `NumPy` array that matches the focal method's expected type, so the test passes without errors. This reveals a key challenge: LLMs tend to imitate the logic of the focal method and produce safe and conventional inputs. Although these inputs are valid, they rarely explore failure-inducing edge cases and are less likely to detect type errors.

To address this limitation, we then prompted the LLM to generate a test explicitly intended to trigger a type error. The resulting `Test 2` successfully triggers an error by passing a `NumPy` array containing a `NaN` value to the method. However, in the real usage of the project, the input is first validated by the method `has_valid_item`, which ensures that each element supports certain required magic methods. Since `NaN` lacks these methods, it violates the context-aware constraint and actually causes a false positive. This motivates the need for extracting context-aware type constraints to guide the test generation process and reduce false positives.

Based on these findings, we further prompted the LLM to generate a test case explicitly aimed at triggering a type error, guided by the type constraint. The resulting `Test 3` shows a common failure mode caused by LLM hallucination. Specifically, the LLM incorrectly assumes that the `TestLocationIndexer` object supports subscript access (i.e., `indexer['x']`), implying the existence of a `__getitem__` method. However, this method is not implemented at all, and thus such access is invalid and results in a `TypeError`. This error does not reflect a fault in the focal method but arises from incorrect assumptions in the generated test. Such hallucinations further introduce false positives, ultimately undermining the reliability of the testing process. This motivates the need for an effective mechanism to identify and refine hallucinated test cases.

To address these challenges, we propose RTED, which enhances LLM-based unit test generation with type constraint analysis and self-reflection for precise type error detection. Specifically, RTED first captures context-aware type constraints via invocation chain analysis and employs an error-seeking agent to propagate constraints backward step-by-step. Then, the constraints are used to guide the test generation process, aiming to detect type errors. Finally, RTED applies a reflection mechanism to iteratively refine test generation, reducing false positives in type error detection. `Test 4` demonstrates the effectiveness of this technique. It tests the `_validate_key` method indirectly via a realistic call chain (`__getitem__` → `_getitem_tuple` → `_has_valid_item` → `_validate_key`) and uses a `datetime` object as input. The input propagates through multiple internal methods, and ultimately exposes the type error in `_validate_key` under the realistic condition.

III. Approach

Figure 2 provides an overview of RTED. Given a focal method and its corresponding invocation chain, RTED proceeds in three main stages. First, in the constraint analysis phase, RTED infers context-aware type constraints. Specifically, an error-seeking agent identifies risky input types and infers the upstream constraints needed to produce them. To avoid unrealistic or hallucinated constraints from the LLM, an evaluation agent verifies their feasibility step by step. If the inferred constraints are plausible and could lead to errors, the invocation chain is marked as high-risk, and the constraints guide bug-oriented test generation. Otherwise, RTED uses a non-error-seeking agent strategy to infer likely correct type constraints to ensure testing sufficiency. Second, in the test generation phase, RTED generates tests for the entry method in the invocation chain, guided by the inferred type constraints and its surrounding context (e.g., class fields, related methods). Finally, in the reflection phase, three specialized agents estimate false positives. A type consistency agent verifies whether the test respects the inferred type constraints. A semantic validity agent checks if the behavior of the test aligns with intended usage. A meta-evaluation agent consolidates these insights and feeds them back to the test generation agent for iterative refinement (if a false positive is estimated). The key steps are explained in detail in the following sections.

A. Constraint Analysis Phase

To solve the problem of LLMs failing to generate type-correct and error-revealing unit tests, RTED enhances LLM-based test generation by explicitly guiding it with type constraints. Specifically, RTED performs a step-by-step type constraint analysis backward along the invocation chain leading to the focal method to infer context-aware constraints likely to trigger type errors. An evaluation agent then estimates the risk of type errors by assessing the feasibility of the inferred constraints at each step along the chain. If the chain is deemed high-risk, the error-revealing constraints are passed to the test generation phase as guidance. Otherwise, RTED falls back to a non-error-seeking agent that infers likely correct input types, ensuring test completeness.

1) Analysis Agents: Let the invocation chain be denoted as $\langle F_1, F_2, \ldots, F_n \rangle$, where F_n is the focal method potentially exhibiting a type error. To assist in detecting such errors and generating valid test inputs, LLMs should be guided by type information. However, due to Python's lack of explicit type annotations and its flexible type system, analyzing F_n in isolation often provides insufficient context and may lead

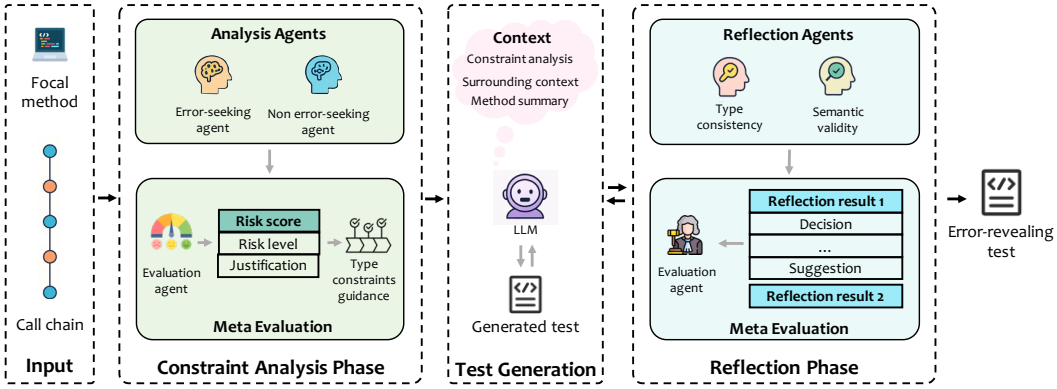

Fig. 2: Overview of RTED

to incorrect or semantically invalid assumptions. Conversely, analyzing the entire chain in one shot is overly complex and prone to hallucinations due to the extended reasoning path.

To this end, RTED performs a step-by-step backward analysis, starting from F_n and proceeding to F_1. Each step analyzes a single function call $\langle F_i, F_{i-1} \rangle$ and infers the type constraint P_{i-1} for F_{i-1}, which describes the expected type constraints of its parameters. However, real-world programs often involve a wide range of types. Representing constraints using concrete types would create an impractically large search space and reduce precision. To mitigate this, RTED represents constraints using a structured schema based on Python primitive types and *magic methods*—special methods (e.g., `__getitem__()`, `__iter__()`) that define behaviors for built-in operations (e.g., subscripting, iteration). Specifically, the constraint associated with each parameter comprises four components:

- **Type**: Indicates whether the parameter is a primitive or a user-defined object.
- **Fields**: Describes the structure and expected type constraints of fields, including nested elements for containers like lists and dictionaries.
- **Custom Methods**: Lists explicitly invoked methods that the parameter should support.
- **Magic Methods**: Lists Python magic methods that the parameter should support.

Notably, RTED represents each constraint in JSON format, with an example illustrate in Figure 3.

Since the type constraints are inferred in a backward manner along the invocation chain, each P_i must be compatible with all downstream constraints P_j where $j > i$. That is, at each step, given $\langle F_i, P_i \rangle$, RTED infers a corresponding constraint P_{i-1} that ensures F_{i-1} can produce inputs satisfying P_i in its call to F_i. This step-by-step inference process can be formally expressed as:

$$\langle F_i, P_i, F_{i-1} \rangle \Rightarrow \langle F_i, P_i, F_{i-1}, P_{i-1} \rangle$$

RTED starts by invoking two specialized agents to infer type constraints in F_n, which serves as the starting point of the

```
{
    "parameter_name": {
        "type": "object",
        "fields": {
            "field_name": {
                "type": "field_type",
                "fields": {...},  // Recursively
                "custom methods": ["method1", "method2"],
                "magic methods": ["__iter__", "__getitem__"]
            }
        },
        "custom methods": ["method1", "method2"],
        "magic methods": ["__iter__", "__getitem__"]
    }
}
```

Fig. 3: An example of type constraints

entire backward propagation process. Each agent focuses on a different aspect of constraint inference:

- Error-seeking agent infers error-triggering constraints P_n^{trigger} likely to expose type errors in F_n.
- Non-error-seeking agent infers valid-use constraints P_n^{normal} that allow F_n to execute successfully.

These complementary constraints help balance fault detection with test completeness. While the error-seeking agent aims to reveal errors, it may occasionally miss bugs or generate unrealistic constraints. In such cases, the non-error-seeking agent offers a fallback by focusing on typical, safe usage patterns. Once F_n is analyzed, RTED continues type constraint inference for each caller in the chain ($\langle F_n \Rightarrow \cdots \Rightarrow F_1$), propagating constraints backward until it reaches F_1.

At the end of the backward inference process, RTED produces two sequences of type constraints: $\langle P_n^{\text{trigger}}, \ldots, P_1^{\text{trigger}} \rangle$ and $\langle P_n^{\text{normal}}, \ldots, P_1^{\text{normal}} \rangle$. These results are then passed to a meta-evaluation phase, where an evaluation agent estimates the likelihood of type errors in the chain and selects the more appropriate constraint set to guide the subsequent test generation process.

2) Meta Evaluation: In the previous step, RTED derived two sets of type constraints: one aimed at revealing errors and another representing correct usage. Then, RTED evaluates the

feasibility and risk of the error-revealing constraints using a dedicated evaluation agent. This agent validates the constraints and estimates the likelihood that the associated invocation chain could trigger a true type error. Specifically, the agent produces two outputs: (1) **Risk level**—labeled as either *high* or *low*, indicating the estimated probability of encountering a type error; (2) **Justification**—a concise explanation summarizing the rationale behind the risk assessment, including which part of the chain contributes most to the potential error.

Ideally, invocation chains with plausible error-revealing constraints are classified as high-risk, and these constraints will be directly used to guide bug-oriented test generation. Otherwise, RTED falls back to the valid-usage constraints inferred by non-error-seeking agent to ensure test completeness.

B. Test Generation Phase

RTED treats *the entry method* of the previously analyzed invocation chain as the method to be tested. This design choice ensures that the generated test cases reflect realistic usage scenarios (i.e., the entry point of a project or module would be invoked in practice). The goal of the test generation phase is to create test cases guided by the type constraints and invocation chain, along with the contextual information of the *entry method*.

1) Context Collection: Prior work on LLM-based test generation has shown that incorporating contextual information beyond the focal method substantially improves the quality of generated tests [5], [14], [15]. Motivated by this insight, RTED constructs comprehensive context to better support test generation. It collects two types of context: *cross-file* and *intra-file* context. The cross-file context includes the full invocation chain analyzed during type constraint analysis, along with the inferred constraints themselves. The intra-file context is gathered by parsing the file hosting the entry method and extracting relevant program elements, specifically including all import statements, global fields, class definitions, and method definitions. If the *entry method* belongs to a class, RTED also collects the class constructor and any class methods that are directly invoked by the *entry method*. The context information is then used to guide the LLM during test generation.

2) Test Generation Process: To guide the LLM with the previously extracted context, RTED begins by inserting the cross-file context, including the invocation chain and its inferred type constraints, into the test generation agent's memory as the chat history. Specifically, each entry (i.e., a round of conversation with the LLM) in this history corresponds to a step from the earlier constraint analysis phase, capturing the type constraint for a single call in the invocation chain. This equips the agent with essential background knowledge on the method's broader usage context. Next, inspired by the success of Chain-of-Thought (CoT) prompting [16], RTED employs a two-stage CoT strategy. In the first stage, it formats the intra-file context to reflect the original source structure, appending the entry method at the end. The LLM is then prompted to summarize the method's functionality according to the context, improving its contextual awareness of the method's semantics. Both the intra-file context (included in the prompt used to instruct the LLM) and the generated functionality summary are also retained in memory as a round of conversation in the chat history. In the second stage, the LLM is prompted to generate unit tests using the stored cross-file context, intra-file context, and method summary. This combination serves as rich guidance, enabling the model to produce more targeted and meaningful test cases.

For each generated test case, RTED first removes all assertions, as they are unnecessary for detecting type errors. RTED then executes the test. If the test fails without raising a type error, RTED invokes a self-debugging step [17], prompting the LLM to revise the test based on the error message and the original code. If the revised test still fails without raising a type error, RTED discards it. Otherwise, if a type error is raised, RTED proceeds to the reflection phase to determine whether the error is a true or false positive.

C. Reflection Phase

Existing researches have pointed out that LLMs are prone to hallucinations, which can lead to the generation of invalid or misleading unit tests [5], [6], [14]. Therefore, when an LLM-generated test case triggers a type error, it is essential to determine whether the error genuinely reflects a type misuse or is simply a result of an ill-formed test (e.g., using an invalid parameter type) to minimize false positives. To this end, RTED employs three specialized agents: two reflection agents (one for type consistency checking and another for semantic validity checking) and a meta-evaluation agent. Together, these agents validate generated failures and iteratively refine the test generation process.

1) Reflection Agents: As discussed in Section I, a valid type error test case must satisfy two conditions: (1) type consistency—the test input should satisfy the inferred type constraints, and (2) semantic validity—the error should arise from a meaningful usage scenario rather than unrealistic scenarios or unrelated logic issues. To enforce these criteria, RTED employs two specialized reflection agents:

- **Type Consistency Agent**: Checks whether the test inputs align with the inferred type constraints.
- **Semantic Validity Agent**: Verifies that the test reflects a realistic use case and that the error genuinely stems from a true type misuse.

Each agent is provided with the inferred type constraints used to generate the test case, the content of the invocation chain, the generated test case, and its execution output, including error messages and stack traces. To compensate for LLMs' limited domain knowledge in diagnosing type-related false positives, RTED adopts a few-shot learning approach. Each agent is guided by two curated examples—one illustrating a true type error and another showing a false positive due to invalid input. Figure 4 presents the examples for the type consistency agent. Due to space limitation, we put other examples used on our project homepage [18].

Both agents produce outputs with four components:

- **Decision**: Classifies the test as a true or false positive.

True positive example

```
# Method under test
def add_strings_and_ints(a, b):
    # Expects both a and b to be of the same type
    return a + b

# Candidate test
def test_add_type_error():
    # a is a string, b is an integer → TypeError
on '+' between str and int
    result = add_strings_and_ints("hello", 5)
```

```
# Failure:
TypeError: can only concatenate str (not "int") to
str

# Why “true TypeError”:
The operation `a + b` itself is invalid given
mismatched types. Both `a ` and `b` satisfies the
type constraints (should support __add__).
```

False positive example

```
# Method under test
def process_list(items):
    if not items:
        raise TypeError:
            # Omitted for brief
    for x in items:
        total += x
    return total

# Candidate test
def test_process_list_type_error():
    # Wrong usage: passing empty list
    result = process_list([])
```

```
# Failure:
TypeError: . . .

# Why NOT the intended TypeError:
The error occurs because `items` is an empty list.
This is a false positive: although empty list is
also iterable, the test violates the type
constraints (“items in the list shuold support
__add__”).
```

Fig. 4: Few-shot examples for type consistency agent

- **Confidence**: Indicates certainty (high, medium, low), which will be used in subsequent meta-evaluation.
- **Rationale**: Summarizes the rationale behind the decision.
- **Suggestions**: If the test is deemed false positive, offers guidance for fixing the test.

The results from the two reflection agents are then forwarded to a third agent, i.e., the evaluation agent, which acts as an arbiter and makes the final decision.

2) Meta Evaluation: The meta-evaluation agent is responsible for aggregating the outputs of the reflection agents to make a final decision. It uses an LLM-based weighted voting strategy that considers structured outputs from the type consistency and semantic validity agents, along with the inferred type constraints and the associated invocation chain. Based on this information, it classifies the test case as either a true positive or a false positive. If the test is deemed a true positive, the test case is retained. Otherwise, the agent synthesizes an explanation along with actionable suggestions based on the rationales from both reflection agents. These suggestions are then fed back to the test generation agent as iterative feedback. This feedback loop allows the test generation agent to refine its output, improving its ability to produce truly bug-revealing test cases while minimizing false positives. If, after refinement, the LLM still does not trigger a true type error, RTED proceeds to test the next method.

IV. Evaluation Design

A. Research Questions

To evaluate the effectiveness of RTED, we formulate the following research questions (RQs).

- **RQ1: To what extent can RTED detect type errors compared to state-of-the-art techniques?** The goal of RTED is to generate unit tests for detecting python type errors, and this RQ investigates its ability in this regard.
- **RQ2: How does each main component of RTED contribute to its overall effectiveness?** This RQ motivates an ablation study to evaluate the individual impact of RTED's core components on its effectiveness.
- **RQ3: Can RTED uncover previously unknown type errors in real-world Python projects?** This RQ assesses the practical applicability of RTED by evaluating its ability to discover new type-related bugs in real-world Python codebases.

B. Subjects

To answer RQ1 and RQ2, we adopted two widely-used benchmarks, i.e., BugsInPy [10] and TypeBugs [11], following prior work [1], [11], [19]. Each benchmark contains a collection of real-world Python type-related bugs along with their corresponding fixed versions. To avoid duplication, we removed overlapping bugs between the two benchmarks following the practice of the existing work [1], [20], [21].

We then attempted to replicate those type-related bugs. However, this replication process presented several challenges. For instance, some required third-party libraries had been removed from PyPI and could no longer be installed (e.g., the `codecov` package required by several bugs in the `core` project[1]). Additionally, many benchmark-listed dependencies conflicted with each other, a problem also noted in various related GitHub issues[2][3][4]. We manually resolved these dependency conflicts to the best of our ability. As a result, we successfully replicated 69 real-world type errors across 16 open-source projects in the two benchmarks, with codebases ranging from 3K to 316K lines of code.

For each bug, we extracted the method responsible for triggering the type error, referred to as the buggy focal method. These methods form the basis for evaluating each technique's effectiveness in detecting Python type errors. We

[1]https://github.com/home-assistant/core/issues/91283
[2]https://github.com/kupl/PyTER/issues/1
[3]https://github.com/kupl/PyTER/issues/2
[4]https://github.com/JohnnyPeng18/TypeFix/issues/1

also extracted the corresponding fixed versions, referred to as non-buggy focal methods, to assess whether a unit test generation technique can avoid producing false positives on correct code. In total, we obtained 138 methods, including 69 buggy ones and 69 non-buggy ones.

To answer RQ3, we applied RTED to the latest versions of six popular open-source Python projects to evaluate its effectiveness in detecting previously unknown type errors. Project selection was guided by three criteria: (1) We excluded repositories that primarily serve as educational resources, tutorials, or textbook materials, as they do not reflect production-level complexity. (2) We included only actively maintained projects with an average commit interval of less than one week. (3) We selected projects with comprehensive documentation and clear setup instructions to ensure compatibility with our experimental environment. We examined top-ranked Python repositories on GitHub (sorted by star count) and retained 50 projects that satisfied all three criteria. Then, considering evaluation costs, we sampled six for this experiment to avoid subjective bias. They are kivy [22], langchain [23], luigi [24], pwntools [25], scipy [26], and scrapy [27]. Note that we did not simply select the top-starred projects, as these are often dominated by AI libraries with highly similar code patterns and error types. Instead, our sampling strategy prioritized domain diversity and evaluation reliability.

C. Compared Techniques

RTED aims to generate effective unit tests for precise type error detection, and thus we adopted both the state-of-the-art type error detection technique and the state-of-the-art LLM-based unit test generation techniques as our baselines:

- **Pyinder** [1]: It is a static type-error detection tool for Python, which incorporates four key features identified through manual investigation for type error detection.
- **CHATTESTER** [5]: The first technique to leverage LLMs for unit test generation. It prompts the LLM with the focal method and its context to generate tests.
- **SymPrompt** [6]: It prompts LLMs to generate one test per execution path of the focal method, aiming to improve code coverage by encouraging path diversity.
- **HITS** [7]: It first leverages LLMs to decompose complex methods into smaller slices and then guides LLMs to generate tests for each slice independently, aiming to improve code coverage.

We did not include the traditional Python unit test generation tool (i.e, Pynguin [4]) for comparison. This is because (1) Pynguin fails to run on many projects in the two benchmarks due to its limited support for Python versions. (2) Many existing studies have demonstrated that LLM-based unit test generation outperforms the traditional Pynguin [6], [15], [28].

D. Measurements

These studied techniques differ in output: test generation techniques produce tests that may trigger runtime type errors, while Pyinder raises static alarms. Therefore, we unify the evaluation by considering a type error as "reported" for a given focal method if a test triggers a type error or a static tool raises an alarm.

1) Outcomes on Buggy Methods: To evaluate the effectiveness in triggering type errors, we ran each test generation technique on the buggy method and executed the generated tests on both the buggy and the corresponding fixed versions. For Pyinder, we applied it to both the buggy and fixed methods. Following the existing definition [29], we categorize the outcomes of each technique as follows:

- True Positive for bug detection (TP_{bug}): The technique reports a type error only on the buggy version, but not on the corresponding fixed version.
- False Positive for bug detection (FP_{bug}): The technique reports a type error on both the buggy and fixed versions.
- False Negative for bug detection (FN_{bug}): The technique fails to report a type error on the buggy version.

Since all target methods under this setting are buggy (i.e., positive samples), there are no true negatives.

2) Outcomes on Non-buggy Methods: To investigate whether RTED can avoid producing false positives on correct code, we ran each technique on the fixed version. The possible outcomes include:

- False Positive on non-buggy methods (FP_{nonbug}): The technique incorrectly reports a type error on the non-buggy version.
- True Negative on non-buggy methods (TN_{nonbug}): The technique correctly does not report any error on the non-buggy version.

All target methods under this setting are non-buggy (i.e., negative samples), and thus there are no true positives and false negatives.

3) Metric Calculation: Based on these outcomes, we measured the effectiveness of each technique using the following metrics:

Accuracy measures the proportion of correct identifications, i.e., detecting type errors in buggy methods while correctly not detecting errors in non-buggy methods. It is calculated as: $\frac{\text{TP}_{bug}+\text{TN}_{nonbug}}{\text{TP}_{bug}+\text{FP}_{bug}+\text{FP}_{nonbug}+\text{TN}_{nonbug}+\text{FN}_{bug}}$.

Precision measures the proportion of true bugs among all samples identified as bugs: $\frac{\text{TP}_{bug}}{\text{TP}_{bug}+\text{FP}_{bug}+\text{FP}_{nonbug}}$.

Recall measures the proportion of true bugs correctly identified out of all true bugs: $\frac{\text{TP}_{bug}}{\text{TP}_{bug}+\text{FN}_{bug}}$.

F1-score is the harmonic mean of Precision and Recall, providing a balanced measure that accounts for both false positives and false negatives: $\frac{2\times\text{Precision}\times\text{Recall}}{\text{Precision}+\text{Recall}}$.

E. Implementation and Environment

We implemented RTED in Python, utilizing Jarvis [30] and tree-sitter [31] for call chain extraction. Jarvis combines flow-sensitive intra-procedural analysis and inter-procedural analysis to infer types and handle dynamic dispatch. Its type inference engine approximates runtime variable types to construct a receiver-type-aware call graph, allowing RTED to retrieve a reasonably sound set of call chains for focal methods. As the underlying LLM, we used DeepSeek-V3 [13]

TABLE I: Comparison among RTED, Pyinder, CHATTESTER, SymPrompt, and HITS

App.	BugsInPy				TypeBugs			
	P	R	F1	Acc	P	R	F1	Acc
Pyinder	0.25	0.11	0.15	0.45	0.14	0.06	0.08	0.44
CHATTESTER	0.25	0.13	0.17	0.50	0.29	0.11	0.16	0.47
SymPrompt	0.09	0.04	0.06	0.43	0.33	0.05	0.09	0.50
HITS	0.11	0.09	0.10	0.40	0.34	0.31	0.33	0.47
RTED	**0.78**	**0.50**	**0.61**	**0.70**	**0.69**	**0.54**	**0.61**	**0.67**

via its API to power all agents within RTED. For Pyinder, we directly leveraged the released implementation [32]. For CHATTESTER (which is originally designed for Java), we adapted it to Python based on its publicly available implementation. For SymPrompt and HITS, due to the lack of released code, we re-implemented them based on the descriptions in their respective papers. To ensure a fair comparison, we used the same DeepSeek-V3 model [13] as the underlying LLM for CHATTESTER, SymPrompt, and HITS. For all focal methods in the benchmarks, we configured each technique to generate one test file per focal method. For RTED, one representative call chain (i.e., the shortest) is sampled per method to guide test generation. This design reduces prompt ambiguity and ensures the input remains within the LLM's context window, since RTED provides all function implementations along the selected call chain as context. This also ensures that the resulting test suites are of comparable scale across techniques, enabling a fair comparison. We executed all generated tests using each project's original testing framework, primarily pytest [33] or unittest [34]. All experiments were conducted on a workstation running Ubuntu 20.04, equipped with a 128-core CPU, 504 GB of RAM, and 4 NVIDIA A800 GPUs.

V. Results and Analysis

A. RQ1: Effectiveness on Type Error Detection

1) Process: For Pyinder, we collected its reported alarms. For test generation approaches, we generated and executed test suites to observe whether any type errors were triggered during runtime. Each technique's output was then mapped to one of several possible outcomes introduced in Section IV-D based on whether it reported a type error for the buggy or fixed version of a method.

2) Results: Table I presents the overall results in terms of precision (denoted as **P**), recall (denoted as **R**), F1-score (denoted as **F1**), and accuracy (denoted as **A**). From the table, RTED consistently outperforms all baselines, including the static analyzer Pyinder and dynamic test generation techniques, across both benchmarks. In terms of accuracy, RTED achieves 0.70 on BugsInPy and 0.67 on TypeBugs, significantly outperforming the best-performing baselines: 0.50 by CHATTESTER on BugsInPy and 0.50 by SymPrompt on TypeBugs, achieving improvements of 40% and 34%, respectively. For F1-score, RTED reaches 0.61 on both datasets, while the best baselines, CHATTESTER (0.17 on BugsInPy) and HITS (0.33 on TypeBugs), fall far behind. This demonstrates that RTED not only detects more bugs but also maintains strong precision, avoiding the excessive false positives that undermine many existing techniques.

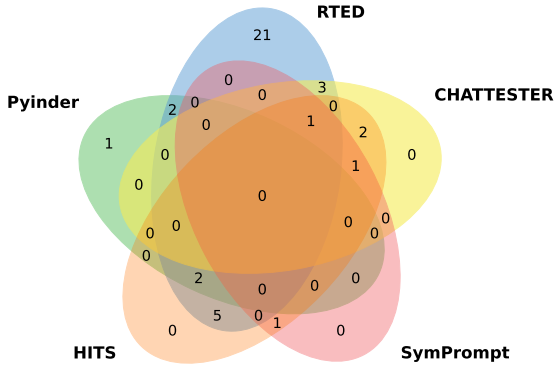

Fig. 5: Overlap of bug detection

Figure 5 shows the overlap in bugs detected by each technique. From the figure, RTED successfully detects 34 bugs, while Pyinder, CHATTESTER, SymPrompt, and HITS detect only 5, 7, 3, and 12 bugs, respectively. Moreover, to detect those bugs, Pyinder, CHATTESTER, SymPrompt, and HITS produces 21, 19, 14, 35 false positives respectively, while RTED only produces 13 false positives, achieving an improvement of 173.9%∼245.9% in precision. Notably, RTED not only covers nearly all bugs found by others but also identifies the largest number of unique bugs with fewest false positives, underscoring its practical effectiveness.

Note that Pyinder, despite being tailored for type error detection, performs worse than the test generation approaches in some cases. This is partly due to its inherent limitations: certain bugs in the benchmarks require runtime information or involve third-party libraries, which static analysis alone cannot effectively handle. For more direct comparison, we also evaluate the performance of Pyinder and RTED on the 40 focal methods supported by Pyinder, including 20 buggy ones and 20 non-buggy ones. On those methods, Pyinder detects only 5 type errors and raises 21 false positives. In contrast, RTED detects 7 bugs while producing just 2 false positives, highlighting its superior recall and precision, both of which are critical for practical usability.

> **RQ1 Summary**: RTED outperforms both static and dynamic baselines in detecting Python type errors, achieving the highest precision, recall, F1-score, and accuracy across two benchmarks. That is, it detects the most bugs, including many unique bugs, while maintaining low false positive rates. These results affirm RTED's effectiveness for Python type error detection.

B. RQ2: Ablation Study

1) Process: In this RQ, we investigate the contributions of the two key components in RTED (e.g., the constraint analysis

TABLE II: Comparison between RTED and its variants

App.	BugsInPy				TypeBugs			
	P	R	F1	Acc	P	R	F1	Acc
RTED$_{w/o\ c}$	0.64	0.25	0.36	0.58	0.58	0.33	0.42	0.62
RTED$_{w/o\ r}$	0.48	0.58	0.53	0.58	0.51	0.59	0.55	0.60
RTED	**0.78**	**0.50**	**0.61**	**0.70**	**0.69**	**0.54**	**0.61**	**0.67**

phase and the reflection phase). To isolate their effects, we construct two variants of RTED:

- **RTED**$_{w/o\ c}$, which removes the constraint analysis component. The LLM generates tests without guidance about type constraints, but still retains the reflection mechanism to iterate based on feedback from test execution.
- **RTED**$_{w/o\ r}$, which removes the reflection component. The LLM is still guided by the results of type constraint analysis to generate unit tests. However, no reflection is applied if a generated test triggers a type error.

We apply both variants to all focal methods using the same setup as RQ1 and analyze their effectiveness.

2) Results: Table II shows that both ablations lead to noticeable performance drops compared to the full RTED, confirming the importance of each component. Specifically, without explicit type constraint guidance, the LLM struggles to generate tests that expose type errors. This is reflected in drastically reduced recall (0.25 on BugsInPy and 0.33 on TypeBugs) and a correspondingly low F1-score (0.36 on BugsInPy and 0.42 on TypeBugs). Despite a somewhat decent precision (0.64 on BugsInPy and 0.58 on TypeBugs), this variant misses a substantial number of actual type errors, indicating that LLMs alone are insufficiently aware of subtle type issues in the absence of type constraints guidance. The lowered accuracy (0.58/0.62) further shows that this variant misidentifies many buggy methods as safe.

In contrast, when reflection is removed, recall remains relatively high (0.58/0.59), showing that initial type-guided prompts are often effective at detecting bugs. However, precision drops sharply (to 0.48/0.51), meaning the tests also trigger many spurious errors. Without reflection, the LLM cannot refine or validate the generated tests, resulting in a higher rate of false positives. The lower accuracy (0.58/0.60) also reflects this unreliability in distinguishing true bugs.

The full version of RTED, combining both type constraints and reflective refinement, achieves the best performance across all metrics. Notably, it achieves the highest F1-score (0.61 on both datasets) and accuracy (0.70/0.67). Compared to RTED$_{w/o\ c}$/RTED$_{w/o\ r}$, it improves F1 by 56%/13%, respectively. This shows that the synergy between the two components is critical: constraint analysis narrows the test generation space toward effective type error detection, while reflection eliminates false positives, enhancing reliability.

TABLE III: Comparison among RTED, Pyinder, CHATTESTER, SymPrompt, and HITS in detecting bugs (**Chat** represents CHATTESTER and **Sym** represents SymPrompt)

Bug	Pyinder	Chat	Sym	HITS	RTED
kivy-1	✓	✓	✗	✓	✓
kivy-2	✗	✗	✗	✓	✓
kivy-3	✓	✗	✗	✗	✓
kivy-4	✗	✗	✗	✗	✓
langchain-1	✗	✗	✗	✗	✓
langchain-2	✗	✗	✗	✗	✓
langchain-3	✗	✗	✓	✓	✓
luigi-1	✗	✗	✗	✗	✓
luigi-2	✗	✓	✓	✓	✓
luigi-3	✓	✓	✓	✓	✓
luigi-4	✓	✗	✗	✗	✓
pwntools-1	✓	✗	✗	✗	✓
scipy-1	✓	✗	✗	✗	✓
scrapy-1	✓	✓	✗	✗	✓
scrapy-2	✗	✗	✗	✓	✓
Total	7	4	3	6	15

RQ2 Summary: Both type constraint analysis and reflection are essential to RTED's effectiveness. Type analysis steers test generation toward likely type errors, while reflection filters out false positives. Their combination enables RTED to achieve a strong balance of precision, recall, and accuracy.

C. RQ3: Detecting New Type Errors

1) Process: In this RQ, we investigate RTED's ability to detect previously unknown type errors in real-world, large-scale Python projects. As introduced in Section IV-B, we selected six open-source Python projects for evaluation. We first used Jarvis [30] to extract call graphs from each project. Then, we identified top-level entry points, and extracted downstream call chains. RTED was then applied to generate and execute unit tests for each of the call chains. For comparison, we applied three baseline test generation techniques to all testable public methods in each project. Pyinder was run on entire codebases as its whole-program analysis requires. All reported errors were manually verified, and potential type errors were reported to developers for confirmation.

2) Results: Table III presents the detailed results. RTED successfully detected 12 previously unknown type errors, outperforming all other techniques. Additionally, three bugs (i.e., kivy-3, langchain-3, and scrapy-2) were identified as duplicates of existing reports filed by other users but had not yet been fixed (indicating already known bugs). This suggests that RTED is effective at discovering bugs that align with realistic usage scenarios encountered by actual users. In comparison, excluding duplicates, Pyinder detected 6 unknown bugs, HITS detected 4, while CHATTESTER and SymPrompt detected only 4 and 1 unknown bugs, respectively. Among the 12 unknown bugs reported by RTED, four have been confirmed or fixed by the developers. This result highlights

TABLE IV: Generalizability of RTED on different LLMs

App.	BugsInPy				TypeBugs			
	P	R	F1	Acc	P	R	F1	Acc
DS	0.78	0.50	0.61	0.70	0.69	0.54	0.61	0.67
QC	0.81	0.46	0.59	0.70	0.81	0.46	0.59	0.69
DS + QC	0.82	0.50	0.62	0.72	0.75	0.57	0.65	0.71

DS represents DeepSeek-V3; QC represents Qwen3-Coder.

RTED's superior capability in exposing type-related issues in real-world codebases.

Listing 1: A simplified version of pwntools-1

```
getattr(input_stream, 'buffer', input_stream).readline
    (_size).rstrip(b'\n')
```

Our manual analysis confirms that RTED effectively detects bugs involving subtle type constraints that are often missed by conventional test generation techniques. As shown in Listing 1, the code calls `rstrip(b"\n")` on the result of `readline()`, assuming it returns a byte string. However, if `readline()` instead returns a regular string, a `TypeError` occurs because `str.rstrip()` does not accept a `bytes` argument. To trigger this bug, the test input must first *satisfy a reachability constraint*: it must implement a `readline()` method whose return value supports `rstrip()`. Otherwise, an `AttributeError` would be raised before reaching the buggy line. Then, the input should not return the expected `bytes` for `readline()`, thereby exposing the type mismatch. RTED is able to detect this issue by first inferring a type that could trigger an error (e.g., `str`). It then propagates backward to supplement additional constraints (i.e., supports `readline()`), and finally uses a `StringIO` object as input to expose the bug. In contrast, all other evaluated test generation tools fail to detect this issue.

RQ3 Summary: RTED demonstrates strong practical effectiveness in detecting previously unknown type errors in actively maintained open-source projects, highlighting its potential to enhance the reliability of modern Python software.

VI. Discussion

Generalizability. To assess generalizability across different LLMs, we evaluated RTED with a different base LLM, Qwen3-Coder (a recent state-of-the-art model) [35], and observed comparable results across benchmarks, confirming RTED's generalizability to LLM choice. Table IV shows the results. Specifically, precision improved slightly while recall declined, likely due to Qwen3-Coder's stronger reasoning, which benefits reflection but less so for constraint analysis. To validate this, we replaced only the reflection agent with Qwen3-Coder, keeping the rest unchanged. This improved performance (F1-score from 0.61 to 0.62/0.65), reinforces that better reasoning models can enhance the reflection phase. This also highlights RTED's flexibility in integrating different LLMs for specialized roles.

Beyond LLM choice, RTED also generalizes across languages and bug types. Its high-level design is language- and error-agnostic, enabling adaptation to other languages and bug types. It follows a constraint-driven framework: given an error type (e.g., type inconsistency), RTED extracts relevant constraints, performs backward analysis along a representative call chain, and generates tests likely to violate those constraints. The key to generalization lies in defining constraint schemas for target errors, which is often straightforward. For example, to detect Java null pointer dereferences, one can specify that certain variables must be non-null before use and propagate this constraint backward to identify potentially violating contexts. The rest of the pipeline remains unchanged. Adapting to a new language mainly involves adjusting prompt formatting and using available call chain extraction tools, which is widely available in other ecosystems (e.g., Soot for Java). Overall, RTED's modular, constraint-centric architecture supports extensibility across languages and error types.

Listing 2: A false-positive produced by RTED

```
def test_request_httprepr(self):
    class HttpRequest:
        def __init__(self):
            self.url = 'http://example.com'
            self.method = 123
            self.headers = None
            self.body = b''
    http_request = HttpRequest()
    request_httprepr(http_request)
```

False-positives produced by RTED. Most false positives in RTED stem from LLM hallucinations during constraint analysis. Listing 2 shows an example. The LLM generated a mock class `HttpRequest` with `self.method = 123`. Passing this object to `request_httprepr()` in Scrapy caused a type error, as the function expects request.method to be a string or bytes. The LLM mistakenly inferred method as an integer since it likely conflated method attributes with nearby numeric HTTP status codes/enumerations. To mitigate such issues, we plan to enhance constraint analysis with lightweight type inference/validation.

VII. Threats to Validity

The **external threat** primarily stems from the generalizability of RTED. To mitigate this, we evaluated it on two established benchmarks, BugsInPy and TypeBugs, which span diverse real-world projects ranging from 3k to 316k lines of code. We also applied RTED to six recent, large-scale open-source projects, uncovering 12 previously unknown bugs. For comparison, we selected state-of-the-art techniques in type error detection and LLM-based test generation. The consistent performance improvements across both benchmarks and unseen projects mitigate this threat to some extent.

The **internal threats** primarily stem from potential implementation errors in RTED or the baselines. To mitigate this, we used the official implementation of Pyinder [32] and adapted CHATTESTER (originally designed for Java) to Python based on its publicly available implementation. For SymPrompt and

HITS, which do not have publicly available code, we reimplemented them according to their original papers and validated their correctness with representative examples. RTED itself underwent rigorous internal testing and review by two authors.

The **construct threats** include LLM randomness, data leakage, and metric selection. To control for randomness, we set the LLM temperature to zero in line with existing work [14], [36]–[38]. To address data leakage, following existing work [3], we checked whether any error-triggering tests generated by RTED matched the reference unit tests in the benchmarks. We found that none of the generated tests aligned with the fixed-version tests. Moreover, all baselines use the same underlying LLM as RTED, so the performance improvements are not due to the data leakage of LLMs. Additionally, RTED successfully detected 12 bugs in recently updated open-source projects, which are not included in the LLM's training data, further mitigating data leakage concerns. Regarding metrics, we used widely-used metrics, Precision, Recall, F1-score, and Accuracy, to measure effectiveness. We also evaluated the efficiency of RTED. The pipeline starts with Jarvis, which constructs call chains in an average of 14.16s. This is followed by type constraint analysis, test generation, and reflection, adding 77.58s per focal method on average, for a total runtime of 91.74s. Compared to baselines, RTED achieves competitive performance: CHATTESTER requires 72.72s, SymPrompt 103.81s, and HITS 153.84s. While these approaches incur similar overheads, they generate substantially more false positives and detect fewer true bugs than RTED. Considering the high manual cost of inspecting false positives, RTED's modest runtime overhead is well justified by its higher precision. Moreover, the process can be further accelerated through parallel execution and optimized LLM inference engines such as vLLM [39].

VIII. Related Work

LLM-based Unit Test Generation. LLM-based test generation approaches can be broadly categorized into two types: training-based and prompting-based [15], [40], [41]. Training-based methods, such as ATHENATEST [42] and A3Test [40], train LLMs on large-scale datasets of unit tests. While these methods have shown strong performance, they require significant computational resources and large amounts of labeled data. In contrast, prompting-based approaches like CHATTESTER [5], SymPrompt [6], HITS [7], and TELPA [15] guide LLMs using contextual prompts, offering more flexibility and reducing reliance on model fine-tuning.

However, these methods primarily focus on improving test coverage and do not explicitly target bug detection. A recent work by Xin et al. introduces an attention-based mechanism to identify defective methods and guide LLMs to generate bug-revealing tests for Java [3]. However, this approach requires a large number of defective method examples with precise error annotations (e.g., faulty lines) for model training, making it costly in terms of time and computational resources. Moreover, such annotated data is difficult to collect in domains like Python type errors. *In contrast to prior approaches that either focus on coverage improvement or rely on supervised training for general bug detection in Java, our method employs type constraint analysis to guide LLMs in generating tests that are more likely to trigger Python type-related bugs, and incorporates a reflection phase to mitigate false positives.*

Python Type Analysis. Several static type analysis tools for Python have been proposed, including Pyre [43], with support for gradual typing and custom annotations; Pyright [2], a fast and feature-rich type checker; Mypy [44], one of the earliest static type checkers in the Python community; and Pytype [45], which does not require explicit type annotations. Recently, Pyinder, which is discussed and compared in our evaluation, builds upon the existing tools to improve type error detection and achieves state-of-the-art performance among static analyzers. *Different from these work,* RTED *takes a dynamic testing approach via step-by-step type constraint analysis and reflection mechanism to iteratively guide the test generation process, enabling more precise detection of type errors.*

Python Type Inference. Static analysis has been extensively applied to infer types in Python programs using techniques such as constraint-based inference [46] and abstract interpretation [47], [48]. More recently, LLM-based methods have emerged for Python type inference. For example, TypeGen [49] combines lightweight static analysis with in-context learning, crafting few-shot Chain-of-Thought (CoT) prompts to enhance type inference performance. TIGER [50] adopts a two-stage generate-then-rank framework to handle Python's complex and diverse type system more effectively. *Unlike these approaches, which aim to infer precise concrete types, typically at the method level. Our approach analyzes type constraints at the call-chain level and represent type constraints in a unified form instead of concrete types.*

IX. Conclusion

In this paper, we present RTED, a novel type-aware unit test generation framework for effectively detecting type errors. RTED combines step-by-step type constraint analysis and reflective validation to guide test generation while minimizing false positives. Our evaluation on two widely-used benchmarks (i.e., BugsInPy and TypeBugs) shows that RTED can detect 22∼29 more benchmarked type errors than state-of-the-art techniques, including Pyinder, CHATTESTER, SymPrompt, and HITS. RTED is also capable of producing fewer false positives, achieving an improvement of 173.9%∼245.9% in precision. Furthermore, RTED detects 12 previously unknown type errors in large-scale real-world Python projects, demonstrating its effectiveness and generalizability in practical application.

Acknowledgment

We thank all the ASE anonymous reviewers for their valuable comments. This work was supported by the National Key Research and Development Program of China (Grant No. 2024YFB4506300), the National Natural Science Foundation of China (Grant Nos. 62322208, 62232001), and the Emerging Frontiers Cultivation Program of Tianjin University Interdisciplinary Center.

REFERENCES

[1] W. Oh and H. Oh, "Towards effective static type-error detection for python," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1808–1820.

[2] "pyright," 2025, https://microsoft.github.io/pyright/#.

[3] X. Yin, C. Ni, X. Xu, and X. Yang, "What you see is what you get: Attention-based self-guided automatic unit test generation," *arXiv preprint arXiv:2412.00828*, 2024.

[4] S. Lukasczyk and G. Fraser, "Pynguin: Automated unit test generation for python," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 168–172.

[5] Z. Yuan, M. Liu, S. Ding, K. Wang, Y. Chen, X. Peng, and Y. Lou, "Evaluating and improving chatgpt for unit test generation," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1703–1726, 2024.

[6] G. Ryan, S. Jain, M. Shang, S. Wang, X. Ma, M. K. Ramanathan, and B. Ray, "Code-aware prompting: A study of coverage-guided test generation in regression setting using llm," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 951–971, 2024.

[7] Z. Wang, K. Liu, G. Li, and Z. Jin, "Hits: High-coverage llm-based unit test generation via method slicing," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1258–1268.

[8] M. Lee, J. Bak, S. Moon, Y.-C. Jhi, and H. Oh, "Effective unit test generation for java null pointer exceptions," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1044–1056.

[9] J. Chen, J. Han, P. Sun, L. Zhang, D. Hao, and L. Zhang, "Compiler bug isolation via effective witness test program generation," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 223–234.

[10] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh *et al.*, "Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies," in *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2020, pp. 1556–1560.

[11] W. Oh and H. Oh, "Pyter: effective program repair for python type errors," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 922–934.

[12] "Pandas," 2025, https://github.com/pandas-dev/pandas.

[13] "deepseek," 2025, https://www.deepseek.com.

[14] L. Yang, C. Yang, S. Gao, W. Wang, B. Wang, Q. Zhu, X. Chu, J. Zhou, G. Liang, Q. Wang *et al.*, "On the evaluation of large language models in unit test generation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1607–1619.

[15] C. Yang, J. Chen, B. Lin, Z. Wang, and J. Zhou, "Advancing code coverage: Incorporating program analysis with large language models," *ACM Transactions on Software Engineering and Methodology*, 2024.

[16] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[17] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," *arXiv preprint arXiv:2304.05128*, 2023.

[18] "rted homepage," 2025, https://github.com/chenyangyc/RTED.

[19] Y. Peng, S. Gao, C. Gao, Y. Huo, and M. Lyu, "Domain knowledge matters: Improving prompts with fix templates for repairing python type errors," in *Proceedings of the 46th ieee/acm international conference on software engineering*, 2024, pp. 1–13.

[20] C. Yang, J. Chen, X. Fan, J. Jiang, and J. Sun, "Silent compiler bug de-duplication via three-dimensional analysis," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 677–689.

[21] J. Chen, Y. Liang, Q. Shen, J. Jiang, and S. Li, "Toward understanding deep learning framework bugs," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–31, 2023.

[22] "kivy," 2025, https://github.com/kivy/kivy.

[23] "langchain," 2025, https://github.com/langchain-ai/langchain.

[24] "luigi," 2025, https://github.com/spotify/luigi.

[25] "pwntools," 2025, https://github.com/Gallopsled/pwntools.

[26] "scipy," 2025, https://github.com/scipy/scipy.

[27] "scrapy," 2025, https://github.com/scrapy/scrapy.

[28] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 919–931.

[29] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "Toga: A neural method for test oracle generation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2130–2141.

[30] K. Huang, Y. Yan, B. Chen, Z. Tao, and X. Peng, "Scalable and precise application-centered call graph construction for python," *arXiv preprint arXiv:2305.05949*, 2023.

[31] "Tree-sitter," 2025, https://tree-sitter.github.io/tree-sitter.

[32] "Pyinder artifact," 2024, https://github.com/kupl/PyinderArtifact.git.

[33] "Pytest," 2025, https://docs.pytest.org/en/stable.

[34] "Unittest," 2025, https://docs.python.org/3/library/unittest.html.

[35] "Qwen3 coder," 2025, https://qwenlm.github.io/blog/qwen3-coder.

[36] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[37] C. Yang, L. Yang, Z. Wang, D. Wang, J. Zhou, and J. Chen, "Clarifying semantics of in-context examples for unit test generation," in *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering*, 2025.

[38] C. Yang, J. Chen, J. Jiang, and Y. Huang, "Dependency-aware code naturalness," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 2355–2377, 2024.

[39] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[40] S. Alagarsamy, C. Tantithamthavorn, and A. Aleti, "A3test: Assertion-augmented automated test case generation," *Information and Software Technology*, vol. 176, p. 107565, 2024.

[41] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to prioritize test programs for compiler testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 700–711.

[42] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers and focal context," *arXiv preprint arXiv:2009.05617*, 2020.

[43] "Pyre," 2025, https://pyre-check.org.

[44] "Mypy," 2025, https://mypy-lang.org.

[45] "Pytype," 2025, https://github.com/google/pytype.

[46] M. Hassan, C. Urban, M. Eilers, and P. Müller, "Maxsmt-based type inference for python 3," in *International Conference on Computer Aided Verification*. Springer, 2018, pp. 12–19.

[47] M. Gorbovitski, Y. A. Liu, S. D. Stoller, T. Rothamel, and T. K. Tekle, "Alias analysis for optimization of dynamic languages," in *Proceedings of the 6th Symposium on Dynamic Languages*, 2010, pp. 27–42.

[48] A. Rigo and S. Pedroni, "Pypy's approach to virtual machine construction," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006, pp. 944–953.

[49] Y. Peng, C. Wang, W. Wang, C. Gao, and M. R. Lyu, "Generative type inference for python," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 988–999.

[50] C. Wang, J. Zhang, Y. Lou, M. Liu, W. Sun, Y. Liu, and X. Peng, "Tiger: A generating-then-ranking framework for practical python type inference," *arXiv preprint arXiv:2407.02095*, 2024.