PyTER: Effective Program Repair for Python Type Errors

Wonseok Oh
Korea University
Republic of Korea
marinelay@korea.ac.kr

Hakjoo Oh*
Korea University
Republic of Korea
hakjoo_oh@korea.ac.kr

ABSTRACT

We present PyTER, an automated program repair (APR) technique for Python type errors. Python developers struggle with type error exceptions that are prevalent and difficult to fix. Despite the importance, however, automatically repairing type errors in dynamically typed languages such as Python has received little attention in the APR community and no existing techniques are readily available for practical use. PyTER is the first technique that is carefully designed to fix diverse type errors in real-world Python applications. To this end, we present a novel APR approach that uses dynamic and static analyses to infer correct and incorrect types of program variables, and leverage their difference to effectively identify faulty locations and patch candidates. We evaluated PyTER on 93 type errors collected from open-source projects. The result shows that PyTER is able to fix 48.4% of them with a precision of 77.6%.

CCS CONCEPTS

• **Software and its engineering** → **Automated static analysis**; **Software testing and debugging**.

KEYWORDS

Program Repair, Program Analysis, Debugging

ACM Reference Format:
Wonseok Oh and Hakjoo Oh. 2022. 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 (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549130

1 INTRODUCTION

Python has become one of the most popular programming languages. According to the IEEE Spectrum's ranking of the top programming languages [1], Python is clearly the dominant language used in a wide variety of applications, including web, enterprise, and embedded domains. In particular, Python's popularity has skyrocketed in recent years, driven by its use in artificial intelligence and data science applications.

*Corresponding author

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

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3549130

Table 1: Top 5 runtime exceptions in Python. We counted the number of StackOverflow posts and GitHub issues (in the repositories used in Section 4) containing the keyword "*Error". Below are the top 5 built-in exceptions (i.e., TypeError, AttributeError, ValueError, KeyError, ImportError) and their relative proportions.

	Type	Attribute	Value	Key	Import
StackOverflow	**31.5%**	19.4%	27.8%	8.3%	13.0%
GitHub	**29.2%**	19.4%	28.2%	12.9%	10.3%

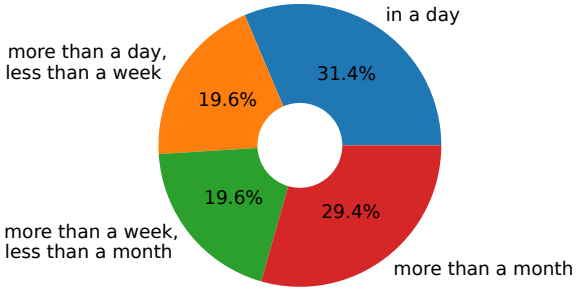


Figure 1: Statistics on the time period between reporting and patching a type error. For bugs in our benchmarks (Section 4), it took 82 days on average and about 30% took more than a month. The worst case took 1,277 days, more than three years.

As a dynamic language, however, Python suffers from an important class of run-time errors, namely type error exceptions. In Python, a type error occurs when an operation is performed on a value of an unsupported type. For example, applying primitive arithmetic operations on an integer and a string (e.g., `1+"two"`) is undefined in Python and hence raises a run-time exception. Python developers struggle with such type errors. Table 1 shows that they appear most frequently among all built-in exceptions in Python. More importantly, type errors are difficult and time-consuming to fix: (1) Correctly fixing a type error requires not only avoiding the given fault, but also anticipating other potential risks in advance (see Section 4.3); (2) In practice, type errors commonly get fixed weeks or months after they are reported (Figure 1).

PyTER. In this paper, we present PyTER[1], the first technique that can automatically fix Python type errors. Despite the importance, no techniques or tools are readily available for fixing type

[1]Python Type Error Repair

```python
def main(x, y, z) :
  if z :
    return foo(x, z, y)
  if y :
    return foo(y, x, z)
  return y

def foo(a, b, c) :
  d = b + c


  e = a + d # TypeError
  return a + c + e
```

(a) Buggy Code

```python
def main(x, y, z) :
  if z :
    return foo(x, z, y)
  if y :
    return foo(y, x, z)
  return y

def foo(a, b, c) :
  d = b + c
  if isinstance(a, str) :
     a = int(a)
  e = a + d # Pass
  return a + c + e
```

(b) Fixed Code

Figure 2: Example to illustrate how PyTER works

errors in dynamically typed languages such as Python. Instead, existing work focused on either detecting type errors in dynamic languages [4, 21, 28, 32, 51] or repairing compile-time type errors in statically typed languages [8, 52]. Also, as our evaluation in Section 4 implies, domain-unaware APR (automated program repair) techniques [33, 41, 43, 47, 57] are unlikely to be effective for real-world type errors.

The key novelty of PyTER is its type-aware APR technique that leverages type information to boost fault localization and patch generation. PyTER aims to fix a type error by handling the incorrect type of a variable with a correct type. To do so, PyTER first collects candidate program variables from error traces and uses dynamic and static analyses to infer the incorrect and correct types that candidate variables may have in erroneous and successful program executions. The difference of those types is then used to accurately localize erroneous program locations and to prioritize candidate patches that are likely to fix the given error correctly.

We prove the effectiveness of PyTER with various type errors in real Python applications. Since there is no benchmark dedicated for Python type errors, we created a new benchmark, called TypeBugs, that includes 93 type errors collected from 15 open-source projects. In total, PyTER successfully fixed 48.4% of those bugs with a precision ($\frac{\#\text{correct patches}}{\#\text{plausible patches}}$) of 77.6%. We also checked that our type-aware APR technique is essential for the performance; the baseline of PyTER, which uses a conventional generate-and-validate approach without our type-aware enhancement, was able to fix 24.7% of 93 bugs with a precision of 48.9%.

Contributions. Our contributions are summarized as follows:

- We present PyTER, the first technique for fixing diverse type errors in real-world Python applications. The key technical contribution is the type-aware APR technique that leverages the different of correct and incorrect types of program variables to enhance fault localization and patch generation.
- We make that the tool and benchmarks publicly available.[2] In particular, we provide a new benchmark for Python type errors, which consists of 15 programs (2.9-428.8 KLoC) and 93 type errors collected from open-source repositories.

[2]https://github.com/kupl/PyTER

2 OVERVIEW

In this section, we illustrate how PyTER works with an example. Figure 2(a) shows a buggy program, where a type error may occur at line 12. Figure 2(b) shows the program fixed by PyTER, where lines 10 and 11 are added to handle the type error.

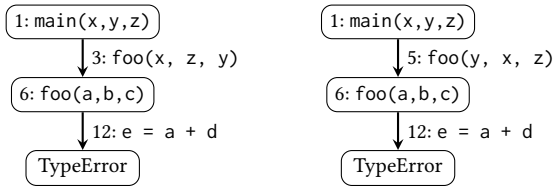
PyTER is a test-based repair technique; it assumes that negative and positive test cases, denoted $\mathcal{T}_N$ and $\mathcal{T}_P$, respectively, are given together with a buggy program. For the example program in Figure 2(a), suppose the following test cases are given:

$$\mathcal{T}_N = \{(("0", 1, 1), 3), ((1, "0", 0), 2)\}, \quad \mathcal{T}_P = \{((0, 0, 0), 0)\}.$$

A negative test, e.g., (("0", 1, 1), 3), consists of an error-triggering input, ("0", 1, 1), to the entry function (`main`) and the corresponding expected output, 3, i.e., the return value of `main`. Note that when the value of `z` is 1, it is considered True at line 2 in Python. Therefore, `foo` is called at line 3 with a string and two integer values being passed to `a`, `b`, and `c`, respectively. Thus, addition at line 12 is performed on unsupported values (i.e., "0"+2) and raises a type error exception. In the second negative test, ((1, "0", 0), 2), the value of `y` is considered as True in the condition statement at line 4. Thus, `foo` is called at line 5 and the same type error exception is raised at line 12. The positive test case, ((0, 0, 0), 0), also consists of an input and an expected output but is used to specify the functionality of the program when it is run successfully without any run-time errors.

Step 1: Collecting Candidate Variables. In Python, type errors typically occur when a variable is used with an incorrect type. Then, developers fix a type error, for example, by converting the erroneous type of the variable to a proper one. In Figure 2(b), the patch at lines 10 and 11 changes the type of variable `a` to integer when it has the string type. Thus, the first step of PyTER is to collect such candidate program variables.

We collect candidate variables from the traceback of the type error exception. A traceback is an error-triggering a call chain that begins with the entry function and ends with a run-time exception. Such a call chain is obtained by running the program with negative test cases. For example, Figure 3 shows the two tracebacks generated from the negative tests in our running example. Given the tracebacks in Figure 3, we collect all program variables that appear in the nodes and edges up to the error location, resulting in *CandVars* = {x, y, z, a, b, c, d} for the program in Figure 2(a) (we do not collect variable `e` that is defined after the type error occurs).



(a) For input ("0", 1, 1) **(b) For input (1, "0", 0)**

Figure 3: Traceback examples

Step 2: Inferring Negative and Positive Types. The second step of PyTER is to infer the types of candidate variables. Specifically, we infer both "negative types" (types that variables have when

	Negative types	Positive types	Difference	Ranking
x	{ int, str }	{ int }	{ str }	2
y	{ int, str }	{ int }	{ str }	2
z	{ int }	{ int }	∅	3
a	{ str }	{ int }	{ str, int }	1
b	{ int }	{ int }	∅	3
c	{ int }	{ int }	∅	3
d	{ int }	{ int }	∅	3

Figure 4: Type analysis example

type errors occur) and "positive types" (types that variables have when type errors do not occur). Figure 4 shows the result of type inference for the program in Figure 2(a). The result shows that variable a, for example, has the string type in the buggy execution, but has the integer type in the normal execution.

We compute negative types using a dynamic analysis that observes variable types while running the program with negative test inputs. For example, when we execute the example program with the inputs ("0", 1, 1) and (1, "0", 0), we find out that variables have types as shown in the second column of Table 4. To obtain positive types, we perform both dynamic and static analyses. Running the program with the positive test input (0, 0, 0) skips the function calls at line 3 and 5, hence we cannot infer the types of variables a, b, c, and d using a dynamic analysis alone. Thus, PyTER makes predictions for those variables using a static analysis specially designed to infer positive types dominantly used in the program (Section 3.2).

Step 3: Fault Localization. PyTER's fault localization is done in two stages. We first perform a function-level fault localization that chooses the function containing the most suspicious candidate variable. We compute the suspicious scores of candidate variables by defining a metric to quantify the difference between negative and positive types. In our example, the most suspicious variable is a because its negative types {str} and positive types {int} are completely different. Thus, we attempt to fix function `foo` first, where variable a is used. Once a function is selected, we identify the most suspicious line by running an existing spectrum-based fault localization (SBFL) technique inside the function. We assume that line 12 is selected for our example program.

We use this two-staged method because existing SBFL is not accurate enough to localize type errors. For example, when we apply SBFL based on $\frac{failed}{failed+passed}$, where *failed* and *passed* denote the numbers of failing and passing test cases, respectively, we have to equally suspect functions `main` and `foo`: lines 3 and 5 of `main` and all lines of `foo` have the same suspicions score. However, it is not possible to fix the bug by repairing `main`.

Step 4: Patch Generation. After fault localization, the following information is available:

$$\delta = (\text{foo}, 12, \text{a}, \{\text{str}\}, \{\text{int}\}))$$

which means that the most suspicious function, line, and candidate variable are `foo`, 12, and a, respectively, and the negative and positive types of a are {str} and {int}, respectively. To fix line 12 of function `foo`, PyTER uses the standard generate-and-validate approach with a set of predefined repair templates designed for type errors and accelerates the procedure by ranking the templates based on the negative and positive types of the candidate variable. In the running example, we note that both {str} and {int} are singleton sets and first try to use a type-casting template that converts the negative type (str) of the candidate variable (a) to the positive type (int), generating the conditional statement at lines 10-12 of Figure 2(b). This patch is accepted by PyTER as it satisfies all test cases in $\mathcal{T}_N$ and $\mathcal{T}_P$.

3 PYTER ALGORITHM

In this section, we describe each step of our approach in detail.

Programs. We consider a small language to describe PyTER, so that its core idea is generally applicable to other languages. A program $P \in Pgm$ is a sequence of function declarations, i.e., $P = F_1, F_2, \ldots, F_n$. A function declaration $F \in FDecl$ is a tuple (f, x, S) of a function name (f), function parameter (x), and body statement (S). We consider statements and expressions below:

$$\begin{array}{rcl} S & \rightarrow & x = E \mid \text{return } E \mid S_1; S_2 \\ E & \rightarrow & n \mid s \mid b \mid x \mid E_1 \oplus E_2 \mid f(E_1, \ldots, E_n) \end{array}$$

where n, s, and b are constant values of integer, string, and boolean types, respectively. We write F_e for the entry function, the starting point of P. We assume program variables are uniquely named and all functions explicitly return a value upon termination. Let Val be the set of values that programs manipulate, e.g., integers, strings. Let $[\![P]\!] : Val \rightarrow Val$ be the evaluator that takes an input value (i.e., an argument value of the entry function) and produces an output value. $[\![P]\!]$ is a partial function and we write $[\![P]\!](v_i) = \perp$ when it is undefined for input v_i (we treat $\perp$ as a special value not included in Val, i.e., $\perp \notin Val$). We assume $[\![P]\!](v_i)$ produces $\perp$ only when it causes a type error. In other words, we assume no other run-time errors can occur for simplicity. Moreover, we assume there is only a single type error in the buggy program.

Problem Definition. Assume a program P with a set $\mathcal{T} \subseteq Val \times Val$ of test cases are given, where a test case is a pair $(v_i, v_o) \in \mathcal{T}$ of input and output values of the program. We assume that for some $(v_i, v_o) \in \mathcal{T}$, $[\![P]\!](v_i) = \perp$. We divide $\mathcal{T}$ into negative tests ($\mathcal{T}_N$) and positive tests ($\mathcal{T}_P$):

$$\mathcal{T} = \mathcal{T}_N \cup \mathcal{T}_P$$

where $\mathcal{T}_N = \{(v_i, v_o) \in \mathcal{T} \mid [\![P]\!](v_i) = \perp\}$ and $\mathcal{T}_P = \{(v_i, v_o) \in \mathcal{T} \mid [\![P]\!](v_i) = v_o\}$. Our goal is then to fix the program by transforming P into P' such that for all $(v_i, v_o) \in \mathcal{T} = \mathcal{T}_N \cup \mathcal{T}_P$, $[\![P']\!](v_i) = v_o$.

3.1 Collecting Candidate Variables

Traceback. We collect candidate variables from the traceback (error trace) that is available in Python when a run-time exception is raised. When $[\![P]\!](v_i) = \perp$ for some error-triggering input v_i, we write traceback($[\![P]\!](v_i)$) for the associated traceback. A traceback T is a sequence of function calls made at an error location:

$$T = \langle (F_1, S_1), (F_2, S_2), \ldots, (F_n, S_n) \rangle$$

where F_i is a function declaration (i.e., $F_i = (f, x, S)$), S_i a statement, and F_1 the entry function (F_e). For $1 \leq i \leq n-1$, S_i is a statement

that includes a function call to F_{i+1}. S_n is the statement where the error occurs.

Candidate Variables. We collect program variables appearing in error traces as candidate variables. We run negative test cases in $\mathcal{T}_N$ to generate the set $\mathbb{T}$ of all tracebacks for the given type error:

$$\mathbb{T} = \{\mathsf{traceback}([\![P]\!](v_i)) \mid (v_i, v_o) \in \mathcal{T}_N\}.$$

Given a traceback T, let $\mathsf{Var}(T)$ be the set of variables appearing in T: $\mathsf{Var}(T) = \bigcup_{((f,x,\_),S)\in T} \mathsf{Var}(S) \cup \{x\}$, where $\mathsf{Var}(S)$ denotes the set of variables used in statement S. We define the set *CandVars* of candidate variables as follows:

$$\mathit{CandVars} = \bigcup_{T\in\mathbb{T}} \mathsf{Var}(T)$$

3.2 Inferring Negative and Positive Types

Next, we infer the negative and positive types of candidate variables. We compute negative types via dynamic analysis by observing the types of variables while running the program with negative test inputs; for all $x \in \mathit{CandVars}$, we compute

$$\mathit{NegTypes}(x) = \{\mathsf{type}(x, P, v_i) \mid (v_i, \_) \in \mathcal{T}_N\}$$

where $\mathsf{type}(x, P, v_i)$ denotes the type that variable x has when P is executed with input v_i (for simplicity, we assume a variable has a single type per program execution).

For positive types, we use both dynamic and static analyses:

$$\mathit{PosTypes}(x) = \mathit{PosTypes}_{dynamic}(x) \cup \mathit{PosTypes}_{static}(x)$$

where $\mathit{PosTypes}_{dynamic}(x)$ and $\mathit{PosTypes}_{static}(x)$ denote the types inferred by dynamic and static analyses, respectively. The former obtained from positive test cases and is defined as follows:

$$\mathit{PosTypes}_{dynamic}(x) = \{\mathsf{type}(x, P, v_i) \mid (v_i, \_) \in \mathcal{T}_P\}.$$

When the given positive test cases are not enough to infer the positive types of some candidate variables (i.e., $\mathit{PosTypes}_{dynamic}(x)$ is undefined for some $x \in \mathit{CandVars}$), PyTER predicts those types using static analysis. However, note that using a conventional type inference algorithm is unlikely to produce useful information in our case, because variables often do not have ground-truth types in dynamic languages. For example, if we perform conventional type inference on the program in Figure 2a, then $\mathit{PosTypes}_{static}$ would include all the possible types (e.g. int, str, bool, etc) for all candidate variables. In particular, it would contain all negative types, i.e., $\forall x \in \mathit{CandVars}.\ \mathit{NegTypes}(x) \subseteq \mathit{PosTypes}_{static}(x)$, which is undesirable because our aim is to leverage the difference of types that variables have in normal and erroneous executions.

To address this issue of standard type inference, we present a new type analysis tailored for finding the *intended positive types* of the program variables. Our type analysis is based on the intuition that, when a type error occurs, a specific buggy variable has an incorrect type while other variables typically have correct types. For example, when we execute program in Figure 2(a) with negative tests, the type (`str`) of the buggy variable `a` is incorrect but other variables such as `b`, `c`, and `d` are of the correct types. Based on this observation, our type analysis infers the intended positive type of each candidate variable $x \in \mathit{CandVars}$ as follows:

(1) We assume x is the buggy variable and initialize a type environment $\Lambda^x_{init} : \mathit{CandVars} \to 2^{\mathit{Types}}$ by assuming all variables but x have the negative types:

$$\Lambda^x_{init} = \lambda y \in \mathit{CandVars}. \begin{cases} \emptyset & \cdots y = x \\ \mathit{NegTypes}(y) & \cdots \text{otherwise} \end{cases}$$

(2) Run a consistency-based type inference algorithm (denoted infer) w.r.t. Λ^x_{init} to predict the intended positive types $\tau^x \subseteq \mathit{Types}$ of x:

$$\tau^x = \mathsf{infer}(x, \Lambda^x_{init}).$$

(3) When the prediction is not deterministic, i.e., $|\tau^x| > 1$, we select one dominantly used in the program. Writing dom_type for this procedure, we can define $\mathit{PosTypes}_{static}(x)$ as follows:

$$\mathit{PosTypes}_{static}(x) = \mathsf{dom\_type}(\tau^x).$$

We apply the above steps for all variables $x \in \mathit{CandVars}$. Now, we explain the second and third steps (infer and dom_type) in detail.

Consistency-based Type Inference. To infer the intended type of a specific variable from the initial type environment, we propose a consistency-based static type analysis.

As typical in type inference, our algorithm begins with generating type constraints. We generate a set $\mathbb{C}$ of constraints from statements in tracebacks:

$$\mathbb{C} = \bigcup_{T\in\mathbb{T}} \bigcup_{((\_,\_,S),\_)\in T} \alpha(S)$$

where α extracts constraints from statements and expressions as follows:[3]

$$\alpha(S) = \begin{cases} \alpha(S_1) \cup \alpha(S_2) & \cdots S = S_1; S_2 \\ \alpha(E) \cup \{(x \doteq E)\} & \cdots S = x = E \\ \alpha(E) & \cdots S = \mathsf{return}\ E \end{cases}$$

$$\alpha(E) = \begin{cases} \alpha(E_1) \cup \alpha(E_2) \cup \{(E_1 \doteq E_2)\} & \cdots E = E_1 \oplus E_2 \\ \bigcup_i \alpha(E_i) & \cdots E = f(E_1, \ldots, E_n) \\ \emptyset & \cdots \text{otherwise} \end{cases}$$

where constraint $(E_1 \doteq E_2)$ indicates that E_1 and E_2 should have equivalent types in order to satisfy type consistency.

Example 3.1. Let us generate constraints from the buggy program in Figure 2a. In case of the statement (e = a + d) at line 12 , we can extract constraints such as:

$$\alpha(\mathsf{e = a + d}) = \alpha(\mathsf{a + d}) \cup \{\mathsf{e} \doteq \mathsf{a + d}\} = \{\mathsf{a} \doteq \mathsf{d}, \mathsf{e} \doteq \mathsf{a + d}\}.$$

The generated constraints for statements in `foo` are as follows:

$$\begin{bmatrix} (\mathsf{d} \doteq \mathsf{b + c}) \\ (\mathsf{b} \doteq \mathsf{c}) \end{bmatrix} \ \boxed{\text{9: d = b + c}} \quad \boxed{\text{12: e = a + d}} \quad \begin{bmatrix} (\mathsf{e} \doteq \mathsf{a + d}) \\ (\mathsf{a} \doteq \mathsf{d}) \end{bmatrix} \quad \begin{bmatrix} (\mathsf{a} \doteq \mathsf{c + e}) \\ (\mathsf{c} \doteq \mathsf{e}) \end{bmatrix} \ \boxed{\text{13: return a + c + e}} \tag{1}$$

After collecting constraints $\mathbb{C}$, we solve them to create the final solution (type environment) $\Lambda : \mathit{CandVars} \to 2^{\mathit{Types}}$. Given a

[3] In practice, we distinguish constraints by their labels (program locations), which is necessary to find dominant types when constraints are not unique.

current solution Λ, the following function Φ updates Λ by iterating over constraints in $\mathbb{C}$:

$$\Phi_{\mathbb{C}}(\Lambda) = \begin{cases} \Phi'_{\mathbb{C}}(\phi(c, \Lambda)), & \cdots \mathbb{C} = c \cup \mathbb{C}' \\ \Lambda & \cdots \mathbb{C} = \emptyset \end{cases}$$

where Φ processes a single constraint of the form $E_1 \doteq E_2$:

$$\begin{aligned} \phi((E_1 \doteq E_2), \Lambda) = & \bigsqcup_{x_1 \in \mathrm{Var}(E_1)} \Lambda[x_1 \mapsto \Lambda(x_1) \cup \sigma_2] \\ & \sqcup \bigsqcup_{x_2 \in \mathrm{Var}(E_2)} \Lambda[x_2 \mapsto \Lambda(x_2) \cup \sigma_1] \end{aligned} \tag{2}$$

where σ_1 and σ_2 are the results of E_1 and E_2 w.r.t. the current solution, i.e., $\Lambda \vdash E_1 : \sigma_1$, $\Lambda \vdash E_2 : \sigma_2$. The operator $\sqcup$ is defined to merge two maps in the pointwise manner. The typing rules are fairly standard such as:

$$\frac{}{\Lambda \vdash n : \{\mathsf{int}\}} \quad \frac{}{\Lambda \vdash s : \{\mathsf{str}\}} \quad \frac{}{\Lambda \vdash b : \{\mathsf{bool}\}} \quad \frac{}{\Lambda \vdash x : \Lambda(x)}$$

$$\frac{f \in \mathit{CastFuns}}{\Lambda \vdash f(E) : \{f\}} \quad \frac{f \notin \mathit{CastFuns}}{\Lambda \vdash f(E_1, \ldots, E_n) : \emptyset} \quad \frac{\Lambda \vdash E_1 : \sigma_1 \quad \Lambda \vdash E_2 : \sigma_2}{\Lambda \vdash E_1 \oplus E_2 : \sigma_1 \cup \sigma_2}$$

where *CastFuns* denotes the set of built-in casting functions (e.g., $\mathit{CastFuns} = \{\mathsf{int}, \mathsf{str}, \ldots\}$)[4]; our analysis is intra-procedural that ignores return values of functions except for built-in casting functions. In (2), we assume that $\Lambda(x_i)$ and σ_i are defined for $i \in \{1, 2\}$. Otherwise, we define $\phi((E_1 \doteq E_2), \Lambda) = \emptyset$.

Given Φ, we can define infer as follows:

$$\mathsf{infer}(x, \Lambda^x_{init}) = (\mathit{fix}_{\Lambda^x_{init}} \Phi_{\mathbb{C}})(x)$$

where $\mathit{fix}_{\Lambda^x_{init}} \Phi_{\mathbb{C}}$ computes the following until a fixed point is reached:

$$\begin{aligned} \Lambda^x_0 &= \Lambda^x_{init} \\ \Lambda^x_i &= \Phi_{\mathbb{C}}(\Lambda^x_{i-1}) \quad (i \geq 1) \end{aligned} \tag{3}$$

Note that the fixed point computation begins with the initial type environment that assumes variables have positive types except for the variable x currently assumed to be buggy.

Example 3.2. To infer the intended type of variable a in Figure 2a, we begin with the following type environment:

$$\Lambda_0 = \Lambda^{-\mathsf{a}} = \{\mathsf{a} \mapsto \emptyset, \mathsf{b} \mapsto \{\mathtt{int}\}, \mathsf{c} \mapsto \{\mathtt{int}\}, \ldots\}$$

Iterating over the constraints in (1) starting from Λ_0 converges to the following:

$$\Lambda = \{\mathsf{a} \mapsto \{\mathtt{int}\}, \mathsf{b} \mapsto \{\mathtt{int}\}, \mathsf{c} \mapsto \{\mathtt{int}\}, \ldots\}$$

from which we conclude that the desired type of a is `int`.

Finding Dominant Types. Let Λ^x be the solution of (3). When the inferred type of x is not deterministic, i.e., $|\Lambda^x(x)| > 1$, our algorithm goes into the next step where we select the type of x dominantly used in the program.

Example 3.3. Consider the following code:

```
1 def main(seq, to_append)
2     if isinstance(to_append, list) :
3         to_append = seq + to_append
4     elif isinstance(to_append, tuple) :
5         to_append += (1,)
6         to_append = seq + to_append # TypeError
7     else :
8         to_append = [seq, to_append]
9 return to_append
```

where we assume seq has a list value and to_append has a tuple value. Then, a type error occurs at line 6 because addition between list and tuple is undefined in Python. By assuming to_append is a buggy variable, our algorithm described so far infers from lines 5 and 8 that the positive types of to_append can be both `tuple` and `list`, respectively, resulting in a non-singleton set {`list`, `tuple`}. In this case, however, we would like to predict the type of to_append as `list` since it is the intent of the developer and to_append is dominantly used as `list` over the program (e.g., lines 3 and 8).

To find out the dominant type, we maintain information $\Omega : \mathit{CandVars} \times \mathit{Types} \to \mathbb{N}$, which counts the number of times types are associated with variables during the fixed point computation. We update Ω whenever the type environment Λ is updated. That is, Ω is updated to the following whenever ϕ is applied in (2):

$$\bigsqcup_{\tau \in \sigma_2} \Omega[(x_1, \tau) \mapsto \Omega(x_1, \tau) + 1] \sqcup \bigsqcup_{\tau \in \sigma_1} \Omega[(x_2, \tau) \mapsto \Omega(x_2, \tau) + 1]$$

where $\Omega(x, \tau)$ is initially 0 for all $x \in \mathit{CandVars}$ and $\tau \in \mathit{Types}$. Let Ω^x be such count information generated when Λ^x is computed. Then, we can define dom_type as follows:

$$\mathsf{dom\_type}(\mathcal{T}^x) = \operatorname*{argmax}_{\tau \in \mathcal{T}^x} \Omega^x(x, \tau).$$

3.3 Type-Aware Fault Localization

PyTER uses the type information to identify the program location to be fixed. Suppose the negative and positive types of candidate variables are given from the previous step:

$$\mathit{NegTypes}, \mathit{PosTypes} : \mathit{CandVars} \to 2^{\mathit{Types}}$$

Function-Level Fault Localization. The main difference from traditional fault localization is that our approach is staged and the most suspicious function is identified first. To do so, we compute the suspicious scores of candidate variables based on the difference of the negative and positive types. We define the score of a variable x as a pair of integers as follows:

$$\mathit{score}(x) = \Big(\sum_{(\tau_1, \tau_2) \in A(x)} -1_{\{\tau_1 = \tau_2\}}, \sum_{(\tau_1, \tau_2) \in A(x)} 1_{\{\tau_1 \neq \tau_2\}} \Big)$$

where $A(x)$ is the cartesian product of $\mathit{NegTypes}(x)$ and $\mathit{PosTypes}(x)$. The first and second elements measure how similar and different $\mathit{NegTypes}(x)$ and $\mathit{PosTypes}(x)$ are, respectively. Here, a smaller score is given when the types are similar, and a higher score is given when the types are different. With *score*, we find out the most suspicious candidate variable $x \in \mathit{CandVars}$:

$$x = \operatorname*{argmax}_{x' \in \mathit{CandVars}} \mathit{score}(x')$$

where we use the lexicographic, total order between scores: $(a, b) > (a', b') \iff a > a' \vee (a = a' \wedge b > b')$. Then we treat the function F as most suspicious that uses the selected variable x.

[4] In Python, type casting is done with functions whose names equal types, e.g., int().

Line-Level Fault Localization. Once the suspicious function F and variable x are chosen, we compute the suspicious scores of program locations in F by running a traditional spectrum-based fault localization (SBFL). We simply define the score of a program location (line) l as follows:

$$score_F(l) = \begin{cases} 1 & \exists T \in \mathbb{T}.\ (F, S_l) \in T \\ \frac{failed(S_l)}{failed(S_l)+passed(S_l)} & \text{otherwise} \end{cases}$$

where S_l represents the statement at line l, $failed(S_l)$ and $passed(S_l)$ denote the numbers of failing and passing test cases that execute statement S_l, respectively. When the function F and the statement S_l appear in some error trace T, we give the highest score. Otherwise, we run SBFL to compute a score based on the numbers of failing and passing test cases.

3.4 Type-Aware Patch Generation

Now we explain the patch generation step of PyTER. We are currently given the suspicious function, line, and variable line as well as the positive and negative types of x:

$$\delta = (F, l, x, NegTypes(x), PosTypes(x)).$$

Let S_l be the suspicious statement at line l.

Repair Templates. We extensively studied developer patches available in open-source programs[5], and concluded that most of the patches for fixing single-variable type errors can be categorized into the 9 templates in Table 2, which are divided into three main categories based on their strategies to fix type errors:

- 'TypeCasting' templates insert a casting statement that converts a negative type of the suspicious variable (x) to a positive type.
- 'Handling' templates handle the type error by replacing the suspicious statement (S_l) or an expression in it by new one.
- 'Guard' templates are used when the suspicious statement (S_l) is a conditional statement. The guard of the statement is strengthened by adding a type check to avoid the error.

A template contains holes ($\square$) of five types: $\square_N$ indicates a negative type in $NegTypes(x)$, $\square_P$ a positive type in $PosTypes(x)$, $\square_C$ a casting expression, $\square_S$ a statement, and $\square_E$ an expression. Table 2 shows examples of buggy statement S_l, template applied to S_l (denoted $S_l^{\square}$) , and candidate patch (denoted S_l^*). Thus, templates with prefix 'Negative' is to check if the suspicious variable (x) has a negative type in $NegTypes(x)$. Likewise, templates with prefix 'Positive' is to check whether the variable x has a positive type in $PosTypes(x)$.

Prioritizing Templates. To fix the bug in S_l, we first need to select a template. Basically, we enumerate all templates, but PyTER prioritizes appropriate ones based on the type information, $NegTypes(x)$ and $PosTypes(x)$ of the suspicious variable (x).

We first decide one of the main categories. We choose the Guard category if S_l is a conditional statement, because it can be only applied when the condition is satisfied. Otherwise, we prioritize either TypeCasting or Handling categories based on the number of positive types (i.e., $|PosTypes(x)|$). We expect $|PosTypes(x)|$ to be one for synthesizing the hole $\square_C$ in TypeCasting. Thus, we select TypeCasting instead of Handling when $|PosTypes(x)| = 1$.

Once a main category is chosen, we prioritize sub categories with the number of negative types (i.e., $|NegTypes(x)|$). It is easy to synthesize $\square_N$, when $|NegTypes(x)| = 1$. Hence, we first select sub templates with prefix 'Negative' when $|NegTypes(x)| = 1$. Otherwise, we choose other sub templates firstly except for templates with prefix 'Negative'. Then, we enumerate the rest of sub categories in increasing size.

Template Instantiation. Once a template ($S_l^{\square}$) is chosen, we need to synthesize holes in it to produce candidate patches. This synthesis procedure is defined by the transition system:

$$(\Theta, \rightsquigarrow, \theta_I, \Theta_F)$$

where Θ is a set of states, $(\rightsquigarrow) \subseteq \Theta \times \Theta$ is a transition relation, Θ_I is a set of initial state, and $\Theta_F \subseteq \Theta$ is a set of final states. A state $\theta \in \Theta$ is a pair $(S^{\square}, \delta)$ of a partial statement with holes and the information δ from fault localization. The initial state $\theta_I \in \Theta_I$ is the pair $(S_l^{\square}, \delta)$ where $S_l^{\square}$ denotes the selected template. The goal of template instantiation is to find a final state $(S_l^*, \_) \in \Theta_F$ that makes the program work correctly:

$$\forall (v_i, v_o) \in \mathcal{T}, [\![P[S_l \mapsto S_l^*]]\!](v_i) = v_o.$$

where $P[S_l \mapsto S_l^*]$ denotes the program P with the statement S_l replaced by S_l^*.

Given $\delta = (F, l, x, NegTypes, PosTypes)$, Figure 5 presents the transition relation for synthesizing holes in partial programs. We write dv_τ for the default value of τ (e.g. $\mathsf{dv}_{\mathsf{int}} = 0$, $\mathsf{dv}_{\mathsf{str}} = ""$, etc). In Python, the expression (isinstance(x, (int, str, bool))) means whether a type of x is one of (int, str, bool). We use this expression when synthesizing $\square_P$. $RetExps(F)$ and $RetTypes(F)$ return a set of the expression of return statements used and the output types of the function F, respectively:

$$\begin{aligned} RetExps(F) &= \{E \mid \mathsf{return}\ E \in RetStmts(F)\} \\ RetTypes(F) &= \{\tau \in \sigma \mid E \in RetTypes(F), PosTypes \vdash E : \sigma\} \end{aligned}$$

where the $RetStmts(F)$ denotes the set of all return statements in F and we use $PosTypes$ as a type environment in $RetTypes$.

3.5 Final Algorithm

Putting it all together, Algorithm 1 shows the final algorithm of PyTER. We first collect candidate variables from error traces (line 2) and perform dynamic and static type analysis (line 3). Let Υ be the result of type analysis phase, which contain information about negative and positive types of candidate variables. By utilizing Υ, we run fault localization to rank candidate faulty locations and iterates over the results (line 4), where δ consists of function (F), line (l), variable (x), negative types ($NegTypes$), and positive types ($PosTypes$). We generate a ranked list of candidate patches (line 5) and apply a patch to the original buggy program (line 6). Next, we execute the candidate patched program on test cases $(\mathcal{T}_P, \mathcal{T}_N)$ at line 7, and obtain failed test cases $\mathcal{T}_N'$. If $\mathcal{T}_N'$ is empty, then we return that candidate program as output. Otherwise, we check if $\mathcal{T}_N'$ is a subset of $\mathcal{T}_N$, which means that some of $\mathcal{T}_N$ passes. Then, we continue to refine the current patch by invoking the algorithm with $\mathcal{T}_N'$ (line 11).

[5]For this study, we used not only our benchmarks, TypeBugs, but also others that do not meet our benchmark selection criteria in Section 4.

Table 2: Repair templates with examples. To illustrate, we assume the suspicious variable is x, $NegTypes(x) = \{\textbf{str}\}$, and $PosTypes(x) = \{\textbf{int}\}$. S_l: buggy statement. $S_l^{\square}$: template for S_l. S_l^*: candidate patch without holes.

Main	Sub	S_l (Buggy stmt)	$S_l^{\square}$ (Template for S_l)	S_l^* (Candidate patch)

Type Casting	Negative TypeCasting	`return x + y`	`if isinstance(x, □N) :`
			`  x = □C`
			`return x + y`
				`if isinstance(x, str) :`
				`  x = int(x)`
				`return x + y`

	Positive TypeCasting	`return x + y`	`if not isinstance(x, □P) :`
			`  x = □C`
			`return x + y`
				`if not isinstance(x, (int)) :`
				`  x = int(x)`
				`return x + y`

	TypeCasting Expression	`return x + y`	`return □C + y`	`return int(x) + y`

Handling	Negative Handling Stmt	`return x + y`	`if isinstance(x, □N) :`
			`  □S`
			`return x + y`
				`if isinstance(x, str) :`
				`  return 0`
				`return x + y`

	Negative Handling Expr	`return x + y`	`return (□E if isinstance(x, □N) else x) + y`	`return (0 if isinstance(x, str) else x) + y`

	Positive Handling	`return x + y`	`if not isinstance(x, □P) :`
			`  □S`
			`return x + y`
				`if not isinstance(x, (int)) :`
				`  return 0`
				`return x + y`

	Exception Handling	`return x + y`	`try :`
			`  return x + y`
			`except :`
			`  □S`
				`try :`
				`  return x + y`
				`except :`
				`  return 0`

Guard	Negative Guard	`if x == 0 :`
		`  return x`
		`else :`
		`  return 0`
			`if not isinstance(x, □N) and x == 0 :`
			`  return x`
			`else :`
			`  return 0`
				`if not isinstance(x, (int)) and x == 0 :`
				`  return x`
				`else :`
				`  return 0`

	Positive Guard	`if x == 0 :`
		`  return x`
		`else :`
		`  return 0`
			`if isinstance(x, □P) and x == 0 :`
			`  return x`
			`else :`
			`  return 0`
				`if isinstance(x, int) and x == 0 :`
				`  return x`
				`else :`
				`  return 0`

$$\frac{\tau \in PosTypes(x)}{\square_S \rightsquigarrow x = \mathrm{dv}_\tau} \quad \frac{\tau \in ReturnTypes(F)}{\square_S \rightsquigarrow \text{return } \mathrm{dv}_\tau} \quad \frac{E \in ReturnExps(F)}{\square_S \rightsquigarrow \text{return } E} \quad \frac{\tau \in PosTypes(x)}{\square_E \rightsquigarrow \mathrm{dv}_\tau} \quad \frac{PosTypes(x) = \{\tau_1, \tau_2, \ldots, \tau_n\}}{\square_P \rightsquigarrow (\tau_1, \tau_2, \ldots, \tau_n)} \quad \frac{\tau \in NegTypes(x)}{\square_N \rightsquigarrow \tau}$$

$$\frac{\tau \in PosTypes(x)}{\square_C \rightsquigarrow \tau(x)} \quad \frac{\forall_{i \in \{1,\ldots,n\}} E_i \rightsquigarrow E_i'}{f(E_1, \ldots, E_n) \rightsquigarrow f(E_1', \ldots, E_n')} \quad \frac{E_1 \rightsquigarrow E_1' \quad E_2 \rightsquigarrow E_2'}{E_1 \oplus E_2 \rightsquigarrow E_1' \oplus E_2'} \quad \frac{E \rightsquigarrow E'}{x = E \rightsquigarrow x = E'} \quad \frac{E \rightsquigarrow E'}{\text{return } E \rightsquigarrow \text{return } E'} \quad \frac{S_1 \rightsquigarrow S_1' \quad S_2 \rightsquigarrow S_2'}{S_1; S_2 \rightsquigarrow S_1'; S_2'}$$

Figure 5: Transition relation for instantiating templates

4 EVALUATION

In this section, we experimentally evaluate PyTER to answer the following research questions:

- **Effectiveness of PyTER**: How effectively can PyTER fix real-world Python type errors?
- **Impact of techniques**: How important are the major techniques used in PyTER?
- **Limitations of PyTER**: What limitations does PyTER have? When is PyTER likely to fail to fix type errors?

4.1 Setup

We implemented PyTER in about 8,000 lines of Python code (v.3.9.1). We implemented dynamic type analysis and fault localization on top of PyAnnotate [27], a framework to instrument Python source code. Our implementation of PyTER supports the full language of Python 3, including object-oriented features and user-defined types. An exception is sub-types; the current implementation of PyTER ignores sub-types because we observed they are rarely used in real-world programs. All experiments were done on a Linux machine (Ubuntu 18.04) with 2 CPUs and 128GB memory, powered by the Intel Zeon Silver 4214 processor.

Benchmarks. We used two benchmark sets: TypeBugs and BugsInPy. We constructed a new benchmark set, TypeBugs, because there is no benchmark dedicated to repairing type errors although datasets for type inference exist [3, 44]. We first collected popular open-source projects from Github that have more than 1000 stars. We then searched for pull requests with the keyword "TypeError" in their messages over the past 5 years (since 2016 up to 2021). Among them, we considered type errors that include both error-triggering test cases and a correct patch written by a

Algorithm 1 The PyTER Algorithm

Input: A buggy program P, test cases $\mathcal{T}_N$ and $\mathcal{T}_P$, tracebacks $\mathbb{T}$
Output: A correct program P' satisfying all test cases in $\mathcal{T}_N \cup \mathcal{T}_P$

```
function REPAIR(P, T_P, T_N, 𝕋)
    C ← CandidateVariables (𝕋)                          ▷ Section 3.1
    ϒ ← TypeAnalysis (P, C, T_P, T_N, 𝕋)                ▷ Section 3.2
    for δ ∈ FaultLocalization(𝕋, ϒ) do                  ▷ Section 3.3
        for S*_l ∈ PatchCandidates(δ) do                ▷ Section 3.4
            P' ← P[S_l ↦ S*_l]                          ▷ Apply the patch
            T'_N ← Execute(P', T_P, T_N)                ▷ T'_N : Failed tests
            if T'_N = ∅ then
                return P'
            else if T'_N ⊂ T_N then
                P* ← REPAIR(P', T_P, T'_N, 𝕋)
                if P* ≠ fail then
                    return P*
    return fail
```

developer. For each bug, TypeBugs includes the repository snapshot of the bug-fixing commit with the developer patch removed. In total, we collected 93 bugs from 15 open-source projects, comprising of small to large scale projects (2.9-428.8 KLoC) on various domains such as machine learning (e.g. scikit-learn), data analysis (e.g., pandas), and scientific computing (e.g. numpy). On average, TypeBugs includes 4.1 negative test cases per bug. We used positive test cases in the file containing the negative test cases, resulting in 91.9 test cases per program on average. We did not use the full set of positive tests provided by each project, because running all of them was prohibitively expensive. For example, the pandas project contains about 170,000 tests and running them takes more than 24 hours.

We also evaluate PyTER on BugsInPy [58], an existing bug benchmark for Python. BugsInPy is a Python version of Defects4J [31], containing 493 bugs of diverse types. Among them, we extracted 57 bugs that raise type error exceptions.

Success Criteria. We manually checked whether the generated patches are correct or not, where a patch is considered correct if it is semantically identical to a developer patch ignoring I/O side effects (e.g., printing a value). We also checked the developer's comments to consider their implicit intention when checking semantic equivalence. In all experiments, we set the time budget for tool execution to 3,600 seconds per bug.

4.2 Results

Results on TypeBugs. Table 3 shows the performance of PyTER on TypeBugs and BugsInPy. To see the effectiveness of PyTER, we compare it with Baseline, the baseline of PyTER that uses a conventional generate-and-validate approach without our type-aware enhancement and static type analysis. More precisely, Baseline uses existing SBFL without our type-aware fault localization and patch prioritization. Baseline uses the same set of templates as PyTER. When synthesizing holes in the templates, Baseline uses *NegTypes* and $PosTypes_{dynamic}$ but it does not use $PosTypes_{static}$ that requires our static analysis technique.

```
1  def foo(out, data) :
2  - out = out + data # TypeError
3  + out = out.encode() + data
4    return out
5
6  def goo(module) :
7    ...
8  - for obj in module.values() :
9  + for obj in module.itervalues() :
10     ...
```

Figure 6: A developer patch example in BugsInPy (simplified from scrpay-30)

In total, PyTER generated 58 plausible patches (which pass all test cases) out of 93 bugs. Among those 58 plausible patches, 45 were correct (equivalent to developer patches), leading to a fix rate of 48.4% ($\frac{45}{93}$) and a precision of 77.6% ($\frac{45}{58}$). On the other hand, Baseline generated 47 plausible patches of which 23 were correct, achieving a fix rate of 24.7% ($\frac{23}{93}$) and a precision of 48.9% ($\frac{23}{47}$). These results confirm that our type-aware APR technique is essential for fixing real-world type errors.

Results on BugsInPy. On average, PyTER fixed 31.6% ($\frac{18}{57}$) of 57 bugs with a precision of 64.3% ($\frac{18}{28}$). By contrast, Baseline was able to fix 14.0% ($\frac{8}{57}$) with a precision of 32.0% ($\frac{8}{25}$).

PyTER is less effective on BugsInPy than TypeBugs (48.4% vs. 31.6%) because BugsInPy was not constructed with type errors in mind. We found that many of those 57 bugs were not originally reported as type errors (e.g., the error reports do not contain the keyword "TypeError"). Therefore, although they raise type error exceptions, their root causes often involve other kinds of bugs as well. Figure 6 shows an example that includes a type error at line 2 but also involves another (non-type) error at line 8. In this case, PyTER succeeds to fix the type error at line 2, but the resulting patch does not pass test cases as the bug at line 8 still remains. This is why we newly created TypeBugs, a collection of bugs originally reported as type errors, to solely evaluate the ability of PyTER for fixing type errors.

Impact of Techniques. PyTER features three new techniques: type-aware fault localization (Section 3.3), type-aware patch prioritization (Section 3.4), and static type analysis (Section 3.2). To evaluate the impact of these techniques, we compared the performance of the following variants of PyTER:

- Baseline: the baseline of PyTER without our fault localization, patch prioritization, and static type analysis.
- Baseline+: Baseline with our patch prioritization
- Baseline++: Baseline+ with our fault localization
- PyTER: Baseline++ with our static type analysis

We evaluate these variants in terms of the number of solved benchmarks, including both TypeBugs and BugsInPy, and the cumulative repair time. Note that, to our knowledge, no existing APR tools are readily available for fixing real-world Python programs (except for those for fixing student programs, e.g., [13, 19, 25]). Thus, we tried to indirectly compare PyTER with general APR tools by including

Table 3: Evaluation Results. #B : the number of bugs for each projects. #G : the number of plausible patches. #C : the number of correct patches. FixRate : ($\frac{\#C}{\#B}$). Prec(Precision) : ($\frac{\#C}{\#G}$). Avg. Time : average running time (sec) per bug.

	Program	#B	Avg. KLoC	NegTest		PosTest		Baseline					PyTER				
				Avg. Num	Avg. Time	Avg. Num	Avg. Time	#G	#C	Fix Rate	Prec	Avg. Time	#G	#C	Fix Rate	Prec	Avg. Time
TypeBugs	airflow	7	73.9	1.0	53.4	13.3	43.9	3	0	0.0%	0.0%	232.0	4	3	42.9%	75.0%	50.5
	beets	1	21.4	1.0	4.0	9.0	7.0	1	0	0.0%	0.0%	8.1	1	0	0.0%	0.0%	3.6
	core	9	230.5	1.4	11.6	23.7	14.7	6	4	44.4%	66.7%	303.7	6	5	55.6%	83.3%	88.7
	kivy	1	44.5	1.0	6.0	4.0	26.0	1	0	0.0%	0.0%	400.57	1	0	0.0%	0.0%	54.3
	luigi	1	13.2	4.0	4.0	1.0	2.0	1	0	0.0%	0.0%	20.6	1	0	0.0%	0.0%	1.5
	numpy	3	53.4	2.0	6.0	72.3	4.7	1	0	0.0%	0.0%	1356.8	2	0	0.0%	0.0%	225.5
	pandas	45	86.9	7.0	42.6	141.1	125.8	21	11	24.4%	52.4%	1478.2	24	19	42.2%	79.2%	1063.4
	rasa	1	45.8	1.0	9.0	7.0	6.0	1	1	100.0%	100.0%	7.3	1	1	100.0%	100.0%	7.3
	requests	4	9.4	1.5	8.0	184.8	33.3	3	1	25.0%	33.3%	942.4	4	4	100.0%	100.0%	167.7
	rich	1	16.1	1.0	9.0	7.0	6.0	0	0	0.0%	n/a	3600.0	0	0	0.0%	n/a	3600.0
	salt	9	364.8	1.0	31.0	12.7	165.6	3	2	22.2%	66.7%	1265.5	6	6	66.7%	100.0%	93.8
	sanic	3	5.6	3.3	5.7	51.0	9.3	1	1	33.3%	50.0%	1406.6	3	3	100.0%	100.0%	43.5
	scikit-learn	5	63.5	1.0	8.0	75.8	13.2	3	2	40.0%	66.7%	1561.6	3	2	40.0%	66.7%	408.7
	tornado	1	12.6	1.0	8.0	171.0	20.0	1	1	100.0%	100.0%	1307.9	1	1	100.0%	100.0%	199.2
	Zappa	2	4.1	1.0	7.0	46.5	748.5	1	0	0.0%	0.0%	1876.3	1	1	50.0%	100.0%	1663.9
	Total	93	112.7	4.1	30.7	91.9	101.2	47	23	24.7%	48.9%	1196.0	58	45	**48.4%**	**77.6%**	651.2
BugsInPy	ansible	1	126.5	1.0	21.0	30.0	13.0	0	0	0.0%	n/a	3600.0	0	0	0.0%	n/a	2549.7
	fastapi	2	7.4	1.0	5.5	7.0	3.0	0	0	0.0%	n/a	10.8	0	0	0.0%	n/a	1.5
	keras	5	26.5	1.0	23.8	14.8	50.4	1	0	0.0%	0.0%	2162.9	1	1	20.0%	100.0%	1769.8
	luigi	7	12.3	1.1	9.3	54.9	23.0	4	1	14.3%	25.0%	207.1	5	3	42.9%	60.0%	15.1
	matplotlib	1	94.9	1.0	8.0	214.0	41.0	0	0	0.0%	n/a	0.0	0	0	0.0%	n/a	0.0
	pandas	25	87.9	9.2	67.4	250.8	130.5	11	4	16.0%	36.4%	2040.3	13	7	28.0%	53.8%	1636.7
	scrapy	10	12.7	1.6	8.7	21.0	6.5	6	2	20.0%	33.3%	62.2	6	5	50.0%	83.3%	12.0
	spacy	1	78.6	1.0	7.0	6.0	7.0	1	0	0.0%	0.0%	9.7	1	0	0.0%	0.0%	79.8
	tornado	2	13.3	1.0	3.5	44.0	3.0	1	0	0.0%	0.0%	1.2	1	1	50.0%	100.0%	0.8
	tqdm	1	1.9	1.0	6.0	54.0	4.0	0	0	0.0%	n/a	0.0	0	0	0.0%	n/a	0.0
	youtube-dl	2	112.0	1.0	9.0	81.5	8.5	1	1	50.0%	100.0%	1815.2	1	1	50.0%	100.0%	1802.9
	Total	57	54.6	4.7	35.7	131.7	67.3	25	8	14.0%	32.0%	1248.4	28	18	**31.6%**	**64.3%**	968.5

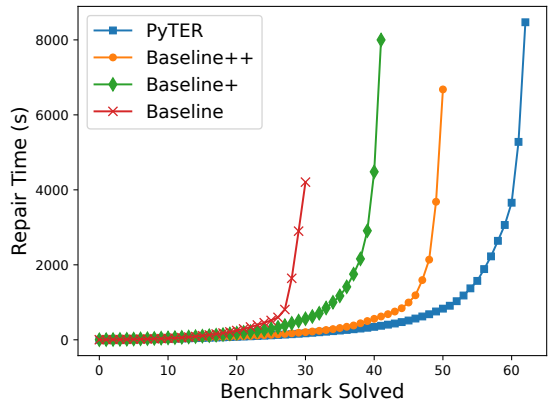


Figure 7: Impact of Techniques

Baseline that can be considered as representing a typical APR tool that is not specifically designed for type errors.

Figure 7 shows the results. The difference between PyTER and Baseline++ shows how useful our static type analysis is. While Baseline++ fixed 51 programs, PyTER fixed 12 more bugs which had insufficient positive test cases. The gap between Baseline++ and Baseline+ shows the impact of our type-aware fault localization; using it increased the number of correct patches from 42 to 51. Moreover, the results show that our patch prioritization technique is also important. Without it, Baseline was only able to repair 31 bugs as opposed to 42 that Baseline+ can fix.

4.3 Discussion

Strength of PyTER over Manual Patches. Because real-world type errors are nontrivial to fix, we found that PyTER often generates patches of higher quality compared to developer patches, which also include all the intended behaviors of the developer patch.

Figure 8 shows an example simplified from scikit-learn. The original code in Figure 8(a) has a type error at line 14. We assume that function `foo` at line 14 only receives string values as its argument and otherwise raises a type error exception. Suppose we are given test cases such that when `main` is called, the value of `s` is False and `k` has a value of the `bytes` type. Then the program executes

```
1  def main(s, k) :
2
3
4    fit = sparse if s else dense
5    fit(k)
6
7  def sparse(k)
8    str_list = ["one", "two"]
9    return str_list.index(k)
10
11 def dense(k)
12
13
14   return foo(k) # TypeError
```

(a) Original code

```
1  def main(s, k) :
2
3
4    fit = sparse if s else dense
5    fit(k)
6
7  def sparse(k)
8    str_list = ["one", "two"]
9    return str_list.index(k) # Future Risk
10
11 def dense(k)
12 + if isinstance(k, bytes) :
13 +   k = k.decode()
14   return foo(k) # Fixed
```

(b) Fix by a developer

```
1  def main(s, k) :
2    fit = sparse if s else dense
3  + if isinstance(k, bytes) :
4  +   k = k.decode()
5    fit(k)
6
7  def sparse(k)
8    str_list = ["one", "two"]
9    return str_list.index(k) # Fixed
10
11 def dense(k)
12
13
14   return foo(k) # Fixed
```

(c) Fix by PyTER

Figure 8: (a) Original buggy code (simplified from scikit-learn-7064). (b) Developer patch. (c) Patch generated by PyTER.

Table 4: Statistics on patch patterns PyTER failed in TypeBugsand BugsInPy.

	TypeBugs (48)	BugsInPy (39)
Group A	10 (21%)	7 (18%)
Group B	14 (29%)	10 (25%)
Group C	7 (15%)	4 (10%)
Group D	5 (10%)	8 (21%)
Others	12 (25%)	10 (26%)

```
1  - user_pass = '%s:%s' % (user, password)
2  + user_pass = to_bytes('%s:%s' % (user, password))
3    creds = b64encode(user_pass).strip() # TypeError
4    ...
5    if creds:
6  -   request = 'Basic ' + creds
7  +   request = b'Basic ' + creds
```

Figure 9: A TypeError bug (line 3) and developer patch (simplified from scrpay-23 in BugsInPy)

function dense and the type error occurs at line 14 because k is not a string value. Figure 8(b) shows the patch written by a developer, where (s)he fixed the type error by inserting the conditional type-casting statement at lines 12 and 13. The developer patch correctly fixed the type error at line 14. However, it is overfitted to the given test cases as a potential error still remains at line 9. When k is a bytes value, the expression str_list.index(k) at line 9 raises an exception (ValueError) because str_list does not contain bytes values. Thanks to our type-aware fault localization, PyTER was able to insert the type-casting statement at lines 3 and 4 and avoided the future risk by always converting the bytes type of k into str before invoking functions sparse or dense at line 5.

Limitations of PyTER. Our evaluation also identified limitations of PyTER. In experiments, PyTER failed to fix 87 bugs. We investigated why PyTER failed and classified those 87 bugs into four groups based on the patterns of desirable patches:

- A : patches that convert incorrect type to correct
- B : patches that need conversion for both types and values
- C : patches that introduce new argument values
- D : patches that insert statements not related to types

The frequencies of these groups are shown in Table 4, where 'Others' are unclassified bugs that require more complicated repair strategies.

Bugs in group A are within the scope of PyTER while other groups are not. So, we further investigated bugs in the A group to see why PyTER failed: 6 bugs needed multi-line patches, 8 bugs required more advanced type inference, 2 bugs required for complex type conversion, and 1 bug required more effective patch prioritization. For example, PyTER failed to produce the patch in Figure 9 that fixes two type errors at lines 1 and 6 at the same time. Currently, PyTER focuses on fixing single-line bugs. Also, to fix the type error at line 1, we need to infer the type of the user-defined function to_bytes. Our static type analysis currently exploits types of built-in casting functions such as int and str.

5 RELATED WORK

APR techniques are commonly classified into special-purpose and general-purpose approaches. Special-purpose techniques aim to fix specific yet important classes of bugs. For example, techniques have been proposed to fix memory errors in C programs [16, 23, 24, 36, 54, 55, 62], null pointer exceptions in Java [12, 37, 60], error-handling bugs [56], concurrency bugs [2, 30, 38, 39], dependency errors [45], and integer/buffer overflow [9, 11, 26, 46, 53]. To our knowledge, PyTER is the first technique specialized for fixing type errors in Python programs. Furthermore, we are not aware of APR techniques applicable to Python except for introductory programming assignments [13, 19, 25].

General purpose approaches [18, 29, 33–35, 40–43, 47, 57, 59, 61] aim to fix any kinds of bugs instead of focusing on specific types of bugs. These techniques are further classified into generate-and-validate [29, 33, 40, 41, 57] and semantic-based approaches [34, 35, 42, 43, 47, 61]. Generate-and-validate approaches explore a predefined search space to generate candidate patches until a patched program which passes the given test cases is found. Semantics-based techniques derive constraints on correct patches and synthesize

patches by using SMT solvers. Technically, PyTER belongs to the generate-and-validate approach, but enhances it with techniques specialized for type errors such as type-aware fault localization and patch prioritization.

A number of analyses have been proposed for type inference in dynamic languages such as Python [7, 17, 21, 49, 50], JavaScript [6, 10, 20, 22, 28, 51], and Ruby [4, 5, 14, 15]. For example, Cannon et al. [7] perform analysis to infer atomic types of local variables. Typepete [21] and DRuby [15] are constraint-based type inference algorithms. These approaches have a limitation; they require user type annotations [7] or assume variables have single types [21]. Neural type inference techniques [49, 50] were also presented, but they need extra learning steps with a large amount of data. TypeDevil [51] and Rubydust [5] dynamically analyze types by running test cases. To address shortcomings of dynamic and static analyses, a hybrid approach [20] has been proposed for JavaScript. Note that our type analysis in Section 3.2 crucially differs from the works described above; our goal is not to infer all the possible types of program variables, which are typically useless for type-error repair, but to predict the intended types when variables are incorrectly used.

6 CONCLUSION

Type errors are common yet difficult-to-fix in dynamic languages such as Python. However, no existing techniques or tools are readily available for use. In this paper, we presented PyTER, the first technique for automatically fixing real-world type errors in Python applications. To this end, we proposed a new APR technique that leverages the difference between negative and positive types of program variables to perform type-aware fault localization and patch generation. Experimental results demonstrated that PyTER can repair diverse type errors effectively; it was able to fix 48.4% of type errors from popular open-source projects with 77.6% precision.

Data Availability. The source code of our tool and benchmarks are available in a public GitHub repository [48].

ACKNOWLEDGMENTS

This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2020-0-01337,(SW STAR LAB) Research on Highly-Practical Automated Software Repair and No.2021-0-00758, Development of Automated Program Repair Technology by Combining Code Analysis and Mining) and the MSIT(Ministry of Science and ICT), Korea, under the ICT Creative Consilience program (IITP-2022-2020-0-01819) supervised by the IITP(Institute for Information & communications Technology Planning & Evaluation), and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No.2021R1A5A1021944).

REFERENCES

[1] [n.d.]. IEEE Spectrum's the Top Programming Languages 2021. https://spectrum.ieee.org/top-programming-languages.

[2] Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. 2017. Repairing event race errors by controlling nondeterminism. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 289–299. https://doi.org/10.1109/ICSE.2017.34

[3] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural Type Hints. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 91–105. https://doi.org/10.1145/3385412.3385997

[4] Jong-hoon An, Avik Chaudhuri, and Jeffrey S Foster. 2009. Static typing for Ruby on Rails. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 590–594.

[5] Jong-hoon An, Avik Chaudhuri, Jeffrey S Foster, and Michael Hicks. 2011. Dynamic inference of static types for Ruby. *ACM SIGPLAN Notices* 46, 1 (2011), 459–472.

[6] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. 2005. Towards type inference for JavaScript. In *European conference on Object-oriented programming*. Springer, 428–452.

[7] Brett Cannon. 2005. *Localized type inference of atomic types in python*. Ph.D. Dissertation. Citeseer.

[8] Sheng Chen and Martin Erwig. 2014. Counter-Factual Typing for Debugging Type Errors. *SIGPLAN Not.* 49, 1 (jan 2014), 583–594. https://doi.org/10.1145/2578855.2535863

[9] Xi Cheng, Min Zhou, Xiaoyu Song, Ming Gu, and Jiaguang Sun. 2017. IntPTI: automatic integer error repair with proper-type inference. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 996–1001. https://doi.org/10.1109/ASE.2017.8115718

[10] Wontae Choi, Satish Chandra, George Necula, and Koushik Sen. 2015. SJS: A type system for JavaScript with fixed object layout. In *International Static Analysis Symposium*. Springer, 181–198.

[11] Zack Coker and Munawar Hafiz. 2013. Program transformations to fix C integers. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 792–801. https://doi.org/10.1109/ICSE.2013.6606625

[12] Thomas Durieux, Benoit Cornu, Lionel Seinturier, and Martin Monperrus. 2017. Dynamic patch generation for null pointer exceptions using metaprogramming. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 349–358. https://doi.org/10.1109/SANER.2017.7884635

[13] Madeline Endres, Georgios Sakkas, Benjamin Cosman, Ranjit Jhala, and Westley Weimer. 2019. InFix: Automatically Repairing Novice Program Inputs. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 399–410. https://doi.org/10.1109/ASE.2019.00045

[14] Michael Furr, Jong-hoon An, and Jeffrey S Foster. 2009. Profile-guided static typing for dynamic scripting languages. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. 283–300.

[15] Michael Furr, Jong-hoon An, Jeffrey S Foster, and Michael Hicks. 2009. Static type inference for Ruby. In *Proceedings of the 2009 ACM symposium on Applied Computing*. 1859–1866.

[16] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. 2015. Safe Memory-Leak Fixing for C Programs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 459–470. https://doi.org/10.1109/ICSE.2015.64

[17] Michael Gorbovitski, Yanhong A Liu, Scott D Stoller, Tom Rothamel, and Tuncay K Tekle. 2010. Alias analysis for optimization of dynamic languages. In *Proceedings of the 6th Symposium on Dynamic Languages*. 27–42.

[18] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Software Eng.* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[19] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2018. Automated Clustering and Program Repair for Introductory Programming Assignments. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 465–480. https://doi.org/10.1145/3192366.3192387

[20] Brian Hackett and Shu-yu Guo. 2012. Fast and precise hybrid type inference for JavaScript. *ACM SIGPLAN Notices* 47, 6 (2012), 239–250.

[21] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. 2018. MaxSMT-based type inference for Python 3. In *International Conference on Computer Aided Verification*. Springer, 12–19.

[22] Phillip Heidegger and Peter Thiemann. 2010. Recency types for analyzing scripting languages. In *European conference on Object-oriented programming*. Springer, 200–224.

[23] David L Heine and Monica S Lam. 2003. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. 168–181.

[24] Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. 2020. SAVER: Scalable, Precise, and Safe Memory-Error Repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE*

'20). Association for Computing Machinery, New York, NY, USA, 271–283. https://doi.org/10.1145/3377811.3380323
[25] Yang Hu, Umair Z. Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. 2019. Re-Factoring Based Program Repair Applied to Programming Assignments. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 388–398. https://doi.org/10.1109/ASE.2019.00044
[26] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. 2019. Using Safety Properties to Generate Vulnerability Patches. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 539–554. https://doi.org/10.1109/SP.2019.00071
[27] Dropbox Inc. 2018. PyAnnotate: Auto-generate PEP-484 annotations. https://github.com/dropbox/pyannotate.
[28] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type analysis for JavaScript. In *International Static Analysis Symposium*. Springer, 238–255.
[29] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 298–309. https://doi.org/10.1145/3213846.3213871
[30] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 389–400. https://doi.org/10.1145/1993498.1993544
[31] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055
[32] Faizan Khan, Boqi Chen, Daniel Varro, and Shane Mcintosh. 2021. An Empirical Study of Type-Related Defects in Python Projects. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3082068
[33] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 802–811. https://doi.org/10.1109/ICSE.2013.6606626
[34] Xuan-Bach Dinh Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. JFIX: semantics-based repair of Java programs via symbolic PathFinder. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, Tevfik Bultan and Koushik Sen (Eds.). ACM, 376–379. https://doi.org/10.1145/3092703.3098225
[35] Xuan-Bach Dinh Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 593–604. https://doi.org/10.1145/3106237.3106309
[36] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2018. MemFix: Static Analysis-Based Repair of Memory Deallocation Errors for C. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 95–106. https://doi.org/10.1145/3236024.3236079
[37] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2022. NPEX: Repairing Java Null Pointer Exceptions without Tests. In *International Conference on Software Engineering*.
[38] Huarui Lin, Zan Wang, Shuang Liu, Jun Sun, Dongdi Zhang, and Guangning Wei. 2018. PFix: fixing concurrency bugs based on memory access patterns. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 589–600. https://doi.org/10.1145/3238147.3238198
[39] Haopeng Liu, Yuxi Chen, and Shan Lu. 2016. Understanding and generating high quality patches for concurrency bugs. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 715–726. https://doi.org/10.1145/2950290.2950309
[40] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 166–178. https://doi.org/10.1145/2786805.2786811
[41] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 298–312. https://doi.org/10.1145/2837614.2837617
[42] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 448–458. https://doi.org/10.1109/ICSE.2015.63
[43] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 691–701. https://doi.org/10.1145/2884781.2884807
[44] Amir M. Mir, Evaldas Latoškinas, and Georgios Gousios. 2021. ManyTypes4Py: A Benchmark Python Dataset for Machine Learning-based Type Inference. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 585–589. https://doi.org/10.1109/MSR52588.2021.00079
[45] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. 2021. Fixing Dependency Errors for Python Build Reproducibility. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 439–451. https://doi.org/10.1145/3460319.3464797
[46] Paul Muntean, Martin Monperrus, Hao Sun, Jens Grossklags, and Claudia Eckert. 2021. IntRepair: Informed Repairing of Integer Overflows. *IEEE Trans. Software Eng.* 47, 10 (2021), 2225–2241. https://doi.org/10.1109/TSE.2019.2946148
[47] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 772–781. https://doi.org/10.1109/ICSE.2013.6606623
[48] Wonseok Oh and Hakjoo Oh. [n.d.]. A replication Package for PyTER: Python TypeError Repair by Type-Aware Generation. https://doi.org/10.6084/m9.figshare.20448573.v1
[49] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. 2022. Static Inference Meets Deep Learning: A Hybrid Type Inference Approach for Python. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2019–2030. https://doi.org/10.1145/3510003.3510038
[50] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: Neural Type Prediction with Search-Based Validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 209–220. https://doi.org/10.1145/3368089.3409715
[51] Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic type inconsistency analysis for JavaScript. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 314–324.
[52] Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, and Ranjit Jhala. 2020. Type Error Feedback via Analytic Program Repair. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 16–30. https://doi.org/10.1145/3385412.3386005
[53] Alex Shaw, Dusten Doggett, and Munawar Hafiz. 2014. Automatically Fixing C Buffer Overflows Using Program Transformations. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*. IEEE Computer Society, 124–135. https://doi.org/10.1109/DSN.2014.25
[54] Tatsuya Sonobe, Kohei Suenaga, and Atsushi Igarashi. 2014. Automatic Memory Management Based on Program Transformation Using Ownership. In *Asian Symposium on Programming Languages and Systems*. Springer, 58–77.
[55] Kohei Suenaga and Naoki Kobayashi. 2009. Fractional ownerships for safe memory deallocation. In *Asian Symposium on Programming Languages and Systems*. Springer, 128–143.
[56] Yuchi Tian and Baishakhi Ray. 2017. Automatically diagnosing and repairing error handling bugs in C. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 752–762. https://doi.org/10.1145/3106237.3106300
[57] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 364–374. https://doi.org/10.1109/ICSE.2009.5070536
[58] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. 2020. 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* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1556–1560. https://doi.org/10.1145/

3368089.3417943

[59] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 416–426. https://doi.org/10.1109/ICSE.2017.45

[60] Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. 2019. VFix: Value-Flow-Guided Precise Program Repair for Null Pointer Dereferences. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 512–523. https://doi.org/10.1109/ICSE.2019.00063

[61] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Trans. Software Eng.* 43, 1 (2017), 34–55. https://doi.org/10.1109/TSE.2016.2560811

[62] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2016. Automated memory leak fixing on value-flow slices for C programs. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*, Sascha Ossowski (Ed.). ACM, 1386–1393. https://doi.org/10.1145/2851613.2851773