Python Predictive Analysis for Bug Detection

Zhaogui Xu*, Peng Liu†, Xiangyu Zhang†, and Baowen Xu*

* State Key Laboratory of Novel Software Technology
* Nanjing University, China
† Department of Computer Science
† Purdue University, USA

zgxu@smail.nju.edu.cn {peng74,xyzhang}@cs.purdue.edu bwxu@nju.edu.cn

ABSTRACT

Python is a popular dynamic language that allows quick software development. However, Python program analysis engines are largely lacking. In this paper, we present a Python predictive analysis. It first collects the trace of an execution, and then encodes the trace and unexecuted branches to symbolic constraints. Symbolic variables are introduced to denote input values, their dynamic types, and attribute sets, to reason about their variations. Solving the constraints identifies bugs and their triggering inputs. Our evaluation shows that the technique is quite effective in analyzing real-world complex programs with a lot of dynamic features and external library calls, due to its sophisticated encoding design based on traces. It identifies 46 bugs from 11 real-world projects, with 16 new bugs. All reported bugs are true positives.

1. INTRODUCTION

Python is becoming one of the most widely used programming languages due to its flexibility, usability, and strong library support for quick development. According to *IEEE Spectrum* [1], the popularity of Python is ranked top 4 in 2015 among all the mainstream programming languages.

Due to the prevalence of Python, analyzing Python programs becomes as important as analyzing programs in traditional languages such as Java and C. However, analyzing Python programs is much more difficult due to the following reasons. First of all, Python is dynamically typed. Variables have no static type declarations and may hold values of different types at runtime. The signatures of Python functions also have no type declarations for the parameters, which allows the functions accepting any types of parameters. Secondly, everything in Python is an object. Particularly, functions, classes and other kinds of objects are all first-class objects that can be created and passed anywhere. Thirdly, object attributes can be dynamically accessed and updated through computed names (e.g., through reflection functions `getattr()` and `setattr()`). Fourthly, collections are pervasively used in Python. They are usually heterogeneous so that they may hold any types of values. Finally, external function calls are pervasive in the Python program.

Existing analysis for Python and other dynamic languages can be classified to three kinds. Static analysis [29, 27, 2, 3] (e.g., abstract interpretation and type inference) analyze program statically. They have limited effectiveness due to the dynamic features in Python. Dynamic analyses [11, 13] focus on the observed executions, and hence cannot analyze unexecuted paths. Symbolic and concolic execution engines [19, 21, 22, 23] are developed for dynamic languages such as Python, Javascript, and Ruby. These engines explore individual paths in a program. Many of them do not support exploring other aspects of the state space such as type variations and attribute set variations that are very common in dynamic languages, due to state explosion and difficulties in explicitly encoding these variations.

In this paper, we develop a predictive analysis engine for Python programs. Predictive analysis [4, 5, 6, 7] is a kind of symbolic analysis driven by concrete executions. It first collects the execution trace of a passing run. It then symbolically encodes the trace, and relaxes the parts that the analyst wants to reason about by introducing symbolic variables to denote possible variations (e.g., schedules in a multi-threaded program execution). Solving the constraints is equivalent to exploring the executions that have small differences from the traced execution to find bugs. We call these the *neighboring executions* of the observed execution.

Our contributions are summarized as follows.

- We develop the first predictive analysis engine for Python programs. It explores neighboring executions of an observed execution by relaxing the inputs, including their types, attribute sets, and values. Different from most predictive analysis that encodes schedule variations and prohibits path changes, it encodes input variations and handles the resulting state changes including path changes. Like other predictive analysis, ours is sound, never reporting false positives.
- We design a novel encoding scheme for Python program and execution trace. It focuses on handling dynamic features by introducing symbolic variables to represent dynamic types and attributes, and explicitly reasoning about their changes and correlations.
- As part of the engine, we develop a tracing infrastructure for Python programs. It features tracing not only the traditional control flow and data flow information, but also the implicit dynamic properties of executed Python statements such as object attributes.
- We build a prototype and evaluate it on 11 real-world Python programs. We have identified totally 46 real bugs, and 16 of them were never found before. The time overhead of our technique is reasonable.

The Essence of Our Technique (Comparison with Other Symbolic Analysis). In general, program state space can be divided into three parts: (1) sub-space that is not explored (e.g., the concretized values in concolic execution); (2) sub-space implicitly explored by the solver (e.g., schedule variations in traditional predictive analysis); and (3) subspace explicitly explored by the engine not the solver (e.g., path exploration in symbolic execution). Different symbolic/concolic analysis differ in the distribution of the three subspaces. For example, traditional symbolic/concolic execution engines (e.g., [22, 19, 20]), including those for dynamic languages, explicitly explore individual program paths. Along each path, they leverage the solver to implicitly explore vari-

able values along the path. Explicit exploration substantially simplifies constraint encoding as only one path is encoded at a time. State variations caused by path variations do not need to be encoded. However, identifying sophisticated exploration strategies and symbolic state checkpointing/backtracking are challenging. Implicit exploration (by the solver) can achieve efficiency leveraging the built-in demand-driven nature of constraint solving in modern solvers. Its difficulties lie in the complexity of constraint encoding.

Compared to existing symbolic/concolic execution, our technique has the following unique features that make it suitable for Python. (1) It encodes values, types, attribute sets, paths, and their correlations so that the solver can be leveraged to perform efficient implicit exploration, whereas most existing techniques explicitly explore paths. Many of them do not reason about type or attribute variations that are critical for dynamic languages. (2) The user provides initial (passing) executions to direct our system to the functionalities/components in which they want to find bugs. For each initial execution, our technique encodes its neighboring state space including paths not executed in the initial run. In other words, one request (to the solver) allows implicit exploration of all the executions in the neighborhood. This completely avoids the highly demanding explicit path exploration. (3) Modeling for the larger number of external functions can be substantially mitigated by trace preprocessing (Section 4.3).

2. MOTIVATION EXAMPLE

In this section, we use a simplified problematic Python program to motivate our approach.

Fig. 1 shows the code snippet simplified from the very popular Python HTTP library `requests`. The test driver function `test_post()` invokes the function `post` at line 276 and checks its return value at line 277. The input parameter `data` can be a dictionary.

Suppose the input parameter `data` is `{"some": "data"}`. The function call `post ()` at line 276 invokes the function `_encode_files()` (lines 87-133) transitively through a chain that consists of six function calls (omitted for simplicity). Line 99 converts the dictionary `data` into a list of pairs, where each pair represents an entry of the dictionary. The resultant list `fields` is `[("some","data")]`. Line 100 iterates over the list and unpacks each pair to `field` and `val`. For the pair `("some","data")`, the unpacked `field` and `val` are `"some"` and `"data"`, respectively. `Val` is wrapped as a list (line 104) if it is not iterable (i.e., without the attribute "*__iter__*") or is of the *str* type (line 101). Here, `val` is wrapped as a list since it is of *str* type. Lines 105-107 then iterate over `val`, form a pair with `field` and store the pair into the list `new_fields`. In this example, `new_fields` is `[("some","data")]`. `New_fields` is then passed to the function `encode_multipart_formdata()` as the parameter `fields`. Lines 42-90 iterate over `fields` and check the `value` in each pair. If it is not of the *int* (line 84) or *unicode* (line 87) type, e.g., it is the string `"data"` in our example, it is written to the `body` (line 90).

The defect is at line 90. The method `body.write()` requires the parameter to implement the buffer interface. For example, the *str* type implements the interface while the *float* type does not. If `body.write()` takes a float value (e.g., 0.0) as parameter, it will crash. Our approach is able to compute the new input `data`, `{"some":0.0}`, to trigger such a crash.

The example illustrates a number of the inherent challenges in analyzing Python programs. Observe that all variables are

```
#Input: url="http://httpbin/post",data={"some":"data"}
267 def test_post(url, data,...):
276   post2 = requests.post(url, data, ...)
277   self.assertEqual(post2.status_code, 200)
   ... # other 6 functions are omitted
# requests/models.py
87  def _encode_files(files, data):
98    new_fields = []
99    fields = to_key_val_list(data or {})
100   for field, val in fields:
101     if not hasattr(val, '__iter__') or
           isinstance(val, str):
104       val = [val]
105     for v in val:
106       if v is not None:
107         new_fields.append((field, v))
133   encode_multipart_formdata(new_fields)
# urllib3/filepost.py
42  def encode_multipart_formdata(fields):
61    body = BytesIO()
65    for fieldname, value in iter_fields(fields):
84      if isinstance(value, int):
85        data = str(value)
87      if isinstance(value, unicode):
88        writer(body).write(value)
89      else:
90        body.write(value)
```

Figure 1: A Motivating Example. (`_encode_files()` is an internal function called during execution whose argument `data` comes from the test input. Unrelated code is omitted for readability.)

dynamically typed such as `value` at line 90. Function parameters can also accept different types. Type errors are only captured by crashes. This example is difficult for static analysis as it involves a lot of container related operations (e.g., the operations in `_encode_files()`) which require complex heap abstraction. This example also imposes challenges to symbolic analysis. For example, the bug manifests in a very deep call chain. The execution has a lot of invocations to external functions, which require substantial efforts to model. In fact, the traced run for the example executes more than 18000 Python statements, among them there are about 2700 container operations and 99 reflection function calls. There are also 946 external functions calls. In addition, we cannot analyze the function `encode_multipart_formdata()` in isolation. In fact, the function is also used in a library `urllib3`, where its use is not buggy since the library performs dynamic checks to ensure the inputs are of the proper types during the function calls.

Our Solution. Our technique belongs to the category of predictive analysis, which starts with the trace of an existing run and predicts a new feasible run. Given the trace collected from the passing run with the input `{"some": "data"}`, our analysis first models the inputs, including their values, types, and attribute sets, as symbolic variables to denote possible input variations; then it encodes the executions possibly derived from the trace into constraints; next it encodes the bug finding assertions, e.g., the `value` is not of the type *str* or *bytearray* (for line 90), which implements the buffer interface; lastly it leverages a solver to compute the input that triggers any assert violation. In this example, we compute the input `data` as `{"some":0.0}`. The predicted run follows the same path as the observed run to reach line 90 and executes `body.write(0.0)`, which leads to the crash.

Our approach has full support of dynamic typing and dynamic attribute updates in Python. The readers may notice that the input contains different types/attributes in the original run and the predicted run (i.e., `{"some": "data"}` and `{"some": 0.0}`), and `value` at line 90 has different type-

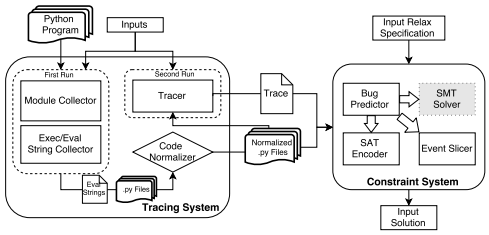

Figure 2: System Framework

s/attributes in the two runs. Our system introduces auxiliary type/attribute symbolic variables to track type/attribute information explicitly and supports type/attribute checks (e.g., lines 84 and 87). In contrast, most existing symbolic execution engines do not reason about type/attribute variations, and hence have difficulties finding the bug.

Our approach not only encodes the current execution path (as in many existing symbolic/concolic execution engines), but also encodes the unexecuted branches together with the path. For example, lines 85 and 88 were not executed in the original run but our technique encodes them with the original run so that the solver can explore these branches implicitly. In constrast, existing symbolic/concolic engines follow some strategy (e.g., depth-first search) to explicitly explore the unexecuted branches through many runs. Details will be discussed in section 5.2.

Besides dynamic types and attribute sets, our approach also supports other dynamic features such as reflection calls and `eval()`, leveraging the traces.

3. SYSTEM FRAMEWORK

In this section, we give an overview of the system. As shown in Fig. 2, it consists of two components: (1) the tracing component and (2) the constraint encoding/solving component.

The tracing component takes a Python program and a test input, and produces a trace of the execution. The tracing procedure consists of three steps. First, the program is run to collect all the involved modules and the string parameters of the `eval()` function invocations, which are loaded or constructed at runtime and are unknown at compile time. This step essentially collects the complete set of executed code. Second, we normalize the code (collected from the modules and the `eval()` parameters) by breaking the complex statements into simple statements. Finally, we run the normalized program again to collect the trace. One prominent challenge in tracing Python program execution is to handle a large number of external function calls that are implemented in other languages and have unknown side-effects. Our tracing component also features an on-the-fly side-effect detection technique to handle external functions.

The constraint encoding/solving component takes three resources as inputs, namely, the trace, the normalized source code and the input relaxation specification[1]. Our system reports bugs and the corresponding inputs that trigger the bugs. The constraint system consists of four sub-components: the bug predictor, the constraint encoder, the event slicer and the SMT solver. The bug predictor subcomponent is a controller that scans the execution trace. During the procedure, it may invoke the other three components. When it finds a satisfiable solution, meaning a bug, it will stop evaluating further. The constraint encoder mainly performs three actions: (1) relaxing the encodings of inputs according to the input specification such that the encoding describes a much larger state space than the traced execution; (2) encoding the execution trace to constraints; (3) aggressively including the unexecuted branches to the constraints whenever encountering a predicate event (in the trace). A trace slicer is used to slice events on which a specific assertion event (i.e., the slice criterion) is dependent. We provide the constraints for the slice to an SMT solver. If it is satisfiable, we find a bug and its triggering input. Otherwise, our tool does not find a violation to the assertion and moves on to the next assertion.

[1]We put the specification design into our technical report for reference: https://sites.google.com/site/pypredictor/

4. TRACING COMPONENT

Similar to existing methods of predictive analysis [4, 5, 8], we also trace the execution based on the 3-address instructions, which are more suitable for constraint encoding than bytecode instructions. Our tracing component follows three steps to collect trace. (1) It executes the program for the first time, and collects all the involved modules and string parameter values for the `exec()/eval()` calls. (2) With the collected modules and string values, it normalizes the source code to a simplified form with each line only having one simple operation. (3) After replacing the original modules with the normalized versions, it re-executes the program and in the meantime collects the execution trace.

4.1 Code Normalization

The official releases of Python have a built-in tracing API `sys.settrace` that allows hooking the execution environment before executing each source code line, through which we can perform a set of queries to the runtime environment, making tracing and gathering runtime information very convenient (e.g., querying values, types and attributes). However, the API only allows tracing at the level of source code line, which is problematic for our purpose as one source code line may contain multiple statements and a long statement may cross multiple lines. As a result, we need to normalize the source code such that each line only contains a simple operation. We mainly perform the following three kinds of normalization.

Linearizing Nested Expressions. Nested Expressions are prevalent in Python code, which combine multiple operations together(e.g., `a.b.f()+x`). We linearize these nested expressions into a group of simple assignments, each containing one simple expression.

Eliminating Complex Constructs. Python has very expressive syntax, with which complex semantics can be represented concisely. While it provides convenience for programmers, it creates difficulties for analysis. Our system transforms these advanced syntactic constructs to a very small set of simple constructs. We list several typical transformations as follows.

`x = [f(i) for i in lst \`
`  if i>0]`
	`x = []`
	`for i in lst:`
	`  if i>0:`
	`  tmp = f(i)`
	`  x.append(tmp)`

`x = reduce(lambda a,b: a+b,\`
`  lst)`
	`def v1(a, b):`
	`  v2 = a + b`
	`  return v2`
	`x = reduce(v1,lst)`

The first row presents a concise method to construct a list by applying a function `f()` to the loop index `i`, with the transformed code on the right. The second row illustrates how we transform a lambda expression.

Transforming `eval()/exec()`. Our system also replaces `eval()/exec()` calls with the corresponding real code collected earlier. We use the following example to illustrate this.

		`co = x + "=a"`
		`if co == "b=a":`
		`  b=a`
`#a.py`	`#records`	`elif co == "c=a":`
`10: co = x + "=a"`	$\langle a\#11, co, b = a\rangle$	`  c=a`
`11: exec(co)`	$\langle a\#11, co, c = a\rangle$	`else:`
		`  exec(co)`

The left column shows the original code that contains an `exec()` call at line 11. The middle column presents the trace records of this line, in which "`b=a`" and "`c=a`" are the values of variable `co` in the two records. The right column shows the corresponding code that our system uses to replace the call. Observe that we use branches to handle the different statements induced by the different values of `co`. We also reserve the original `exec(co)` in the else branch to ensure soundness. With the transformed code, we now can trace the execution of the original `execi()` call.

4.2 Trace Generation

After replacing the involved modules with the normalized modules, we now re-execute the program to gather the execution trace. Tracing Python program execution is more challenging than tracing programs in many other languages. This is because the execution of a Python statement may have many implicit side-effects, such as type changes and attribute set changes. All these need to be explicitly captured as many bugs are related to these implicit side-effects. In the following, we first introduce some notations and then illustrate the tracing rules.

Fig. 3 presents the definitions related to the tracing rules. Basically, v represents an arbitrary value. Note that every value in Python is essentially an object, so the value here means an object; A represents an attribute set which consists of attributes; T represents all the types as well as the class hierarchy of these types ($\preccurlyeq$ denotes the subtype relation); σ denotes the value store mapping a variable or a heap location to a value; ρ represents a type store mapping a variable or a heap location to a type; π denotes an attribute set store mapping a value to an attribute set, representing the set of attributes of a value (or object); $\mathcal{F}_{baset}$ represents a function returning the default attribute set for a specific type; $\mathcal{F}_{btype}$ returns the type of a binary operation; $\mathcal{F}_{index}$ returns the index of the next iteration of a for-loop. The execution of a Python program in our system is a procedure of dynamically updating T, σ, ρ and π.

Trace and Events. We model a few Python statements in our presentation shown in Fig. 4. Some of the symbols used in Fig. 4 are defined in Fig. 3. For example, ℓ stands for a label that is similar to the source code line number. Observe that we model two specific predicates: the type check ($isinst()$) and the attribute check ($hasattr()$). The rest is standard.

An execution trace Γ is a sequence of events recorded during the execution. An event is an execution instance of a statement, denoted as a tuple $\langle eid, s, (v, \tau, A), \ell\rangle$ where eid represents a unique identifier; s denotes the statement; (v, τ, A) represents the involved value, type and attribute set by the execution of the statement. Note that besides the value v, we also record the type and attribute set of each execution of the statement as they may be changed during the execution.

Tracing Rules. Table 1 represents the tracing rules for a set of commonly used Python statements. The first four rules

$v \in Value$	$\tau \in Type$	$x \in Var$	$\ell \in Label$
$l \in HeapLoc$	$::=$	$\overline{Value \times Attr}$	
$A \in AttrSet$	$::=$	$\overline{Attr}$	
$T \subseteq TypeSet$	$::=$	$\overline{(\mathcal{P}(Type), \preccurlyeq)}$	
$\Gamma \in Trace$	$::=$	$\overline{\langle eid, s, (v, \tau, A), \ell\rangle}$	
$\sigma \in ValueStore$	$::=$	$Var \vert HeapLoc \mapsto Value$	
$\rho \in TypeStore$	$::=$	$Var \vert HeapLoc \mapsto Type$	
$\pi \in AttrSetStore$	$::=$	$Value \mapsto AttrSet$	

$\mathcal{F}_{baset}(\tau)$ returns the default attribute set of type τ;
$\mathcal{F}_{btype}(\tau_1, \tau_2, bop)$ returns the type for the computation on τ_1 bop τ_2.
$\mathcal{F}_{index}(v)$ returns the index of next iteration for a for-loop.

Figure 3: Definitions related to the tracing rules

Statement $s ::=$	
$z =^{\ell} v \vert x \vert (x\ op\ y)$	**(basic assignments)**
$y =^{\ell} x.f$	**(attr-read)**
$y.f =^{\ell} x \vert v$	**(attr-write)**
$\texttt{if}\ (x\ relop\ y)^{\ell}\ s_1 \texttt{else}\ s_2$	**(cmp-predicate)**
$\texttt{if}\ isinst(x,t)^{\ell}\ s_1 \texttt{else}\ s_2$	**(isinst-predicate)**
$\texttt{if}\ hasattr(x,f)^{\ell}\ s_1 \texttt{else}\ s_2$	**(hasattr-predicate)**
$y = f^{\ell}(a_1, ..., a_n) \vert x.m^{\ell}(a_1, ..., a_n)$	**(callsite)**

Figure 4: Python Statements

specify four typical object creation statements. More specifically, rule (`class-def`) represents the tracing of a class definition statement, which creates and initializes a new class object. It creates a class object o_1 and associates it with C. It also creates a method object o_2, associated with the m attribute of o_1. The attribute "`m`" is added to the attribute set of the class object o_1. *MethodType* and *Classtype* are two built-in types in Python. *CType* and *BType* are symbols generated by our system to denote the types of class C and B. As shown in the last column, the system generates two events e_1 and e_2 where e_1 represents the class object creation and e_2 represents the setup of the method in the class.

Rule (`func-def`) specifies the rule for a function definition. It creates a function object and then assigns it to variable f. The corresponding trace event is straightforward. Rule (`list-alloc`) represents the tracing for creating a list object. It first creates a list object o_1 assigned to y and then appends two elements to it (o_1_0 represents the first element). Accordingly, our system generates three events e_1, e_2 and e_3 with e_1 creating the list object, and e_2 and e_3 setting the elements. Rule (class-instant) represents the object creation through class instantiation. It creates a new object o_1 and assigns it to y. The attribute set of o_1 is inherited from the object associated with the class C.

Rules (`simple-assign`), (`bop-assign`), (`attr-read`) and (`attr-write`) are standard. Particularly, the execution of an attribute write statement may dynamically change the attribute set of the object.

Rule (`for-loop`) specifies the tracing for the next-iteration operation of a for-loop statement. The rule only specifies the situation in which the object is a collection (e.g., list, tuple). Rule (`if-cmp`), (`if-isinst`) and (`if-hasattr`) specify the tracing rules of three different kinds of branch predicates. All of them are straightforward.

In Rule (`call`), we record two events, one for parameter passing and the other for the call itself.

4.3 Handling External Functions

Function calls to external libraries are prevalent in Python programs. We cannot easily trace the execution of these libraries due to the lack of source code. However, if we do not trace their execution, the resulting trace is incomplete especially when these functions have side-effects on heap, leading to unsoundness of our later symbolic analysis. One way is to manually model the external functions. However, this re-

quires tremendous efforts due to the sheer volume of these functions. Hence, our strategy is to manually model the commonly used external functions. For the others (which are still in a large number), we develop a technique to detect side-effects such that we handle them to avoid unsoundness. Next, we focus on discussing side-effect detection.

To detect side-effects, we maintain a value store $\theta : Var|HeapLoc \mapsto Value$ recording the current value of each variable. The value store is only updated when the Python code is executed. In other words, it is not updated by execution of external functions. When a variable is used in the Python code, we compare the value from the store θ with the observed value from store σ. If the two are different, there must be side-effects by some external function call(s).

`1: x.f = 1`
`2: y=extfunc(x)`
`3: z = x.f`
	$v_e \leftarrow \sigma[\sigma[x].f]$
	$v_r \leftarrow \theta[\theta[x].f]$ `#here,` v_r`=1`
	$if\ \ v_e = v_r:$
	$\quad \Gamma \rightarrow \Gamma; \langle z = x.f, (1, ...)\rangle$
	$else:$
	$\quad \Gamma \rightarrow \Gamma; \langle x.f = 1, (1, ...)\rangle$
	$\quad \Gamma \rightarrow \Gamma; \langle z = v_e, (v_e, ...)\rangle$
	$\theta \rightarrow \theta[z \mapsto v_e]$

Line 2 of the left column shows an unmodeled external function call, which has side-effects on `x`. When tracing the execution of line 3, we use the rule shown in the right column. We first get the current value v_e of the heap location pointed-to by $x.f$ from the execution environment (σ), which is updated by both the Python code and the external function, and then query the recorded value of $x.f$ from θ, which is only updated by Python code. If v_e is equal to 1, we treat it as an attribute read event. Otherwise, we concretize both the values of $x.f$ and z due to the invisible side-effects that we cannot model, which essentially under-approximates the program state by fixing the input and output values of the external function. On the other hand, it ensures soundness.

5. CONSTRAINT ENCODING

Our technique is essentially a kind of predictive analysis [4, 5, 6]. As mentioned in Section 1, predictive analysis is based on the observed execution traces and can easily analyze the very long executions similar to the observed traces. Leveraging traces also allows mitigating the problems caused by dynamic features. While most existing predictive analyses explore different schedules of a concurrent execution and prohibit path changes, our technique explores the program state space caused by input variations, which may cause execution path differences and heap state differences. Such possible differences are encoded as constraints and an SMT solver is leveraged to explore the encoded state space to find bugs. Our technique is sound, meaning that it does not produce false positives. Next, we introduce the set of main challenges in encoding Python programs for predictive analysis.

(1) Dynamic typing in Python makes the type of a variable path sensitive, which means the same variable at the same program point may have different types depending on the execution path. However, symbolic variables in constraints are statically typed. Existing static encoding techniques often project a program variable (in its SSA form) to a symbolic variable in the constraint system. Such a method is no longer applicable. Note that the SSA form does not help in our context because even though the SSA form can express a variable having different types at different locations (by introducing multiple variables for each definition location), it cannot handle that a variable may have different types even at the same location. Note that the problem does not exist for symbolic execution or concolic execution as these techniques encode one path at a time.

Source Code	Trace
`1: x = input()`	$\langle x = 1, (1, int, A_{int})\rangle$
`2: y = C() #new object o1`	$\langle y = o1, (o_1, C, A_{o_1})\rangle$
`3: if isinstance(x, int):`	$\langle if\ isinst(x, int), ...\rangle$
`4:  y.f = "S"`	$\langle y.f = \text{“}S\text{”}, (\text{“}S\text{”}, str, A_{str})\rangle$
`5: else:`	
`6:  y.f = 2`	
`7: z = y.f`	$\langle z = y.f, (\text{“}S\text{”}, str, A_{str})\rangle$
`8: w = z.lower()`	$\langle w = z.lower(), ...\rangle$

Figure 5: An Example Illustrating the Challenges

(2) Besides encoding the value of a variable like in most existing work, we have to explicitly encode the type and the attribute set of a variable (or object) in order to reason about such features.

(3) Some bugs can only be triggered when the execution goes through a path different from the trace. Therefore, to predict such kind of bugs, we should also include the unexecuted branches into the constraint model.

Example. We use an example in Fig. 5 to illustrate the above challenges. The left column is the code snippet. Line 1 represents the initialization of `x` which may accept a value of various types (e.g., int and string). Line 2 creates a new object of class `C`, denoted as `o1`. Line 3 is a type check predicate. If `x` is of the int type, it writes a string “`S`” to the attribute “`f`” of `o1`. Otherwise it writes the integer 2. Note that, the initial attribute set of `o1` does not contain the attribute “`f`”, so the execution will dynamically add it to the attribute set of `o1`. Line 7 reads the attribute and assigns it to `z`. Line 8 calls a method `lower()` of z. The bug will occur when the else branch is taken because the value of `z` is of the int type and does not have the `lower()` method. However, the bug does not manifests itself when the true branch is executed.

From the example, we observe that `z` at line 7 may have different types depending on the path and SSA cannot address the problem. In order to reason about the predicate at line 3 and the bug at line 8, we need to explicitly encode the types and attribute sets. Assuming the trace is along the true branch, in order to expose the bug, we also need to encode the else branch, which may not be taken in the observed execution.

5.1 Encoding Scheme

Next we explain the encoding scheme from different aspects.

Using Multiple Symbolic Variables to Denote the Value of a Python Variable. We use multiple symbolic variables (of different types) to represent the value of a Python variable of different types. More specifically, given a Python variable x, we use a set of symbolic variables in the form of ψ^{σ}_{τ} to denote its value of type τ. The superscript indicates this is a symbolic variable for *value*. For example, if x can be an integer or a string, we use two symbolic variables $\psi^{\sigma}_{int}(x)$ and $\psi^{\sigma}_{str}(x)$ to denote its value. For a statement s involving x, both symbolic variables are updated to encode the possible behavior of s when x has different types.

Explicitly Reasoning about Types. To encode the dynamic typing behavior, we introduce a new symbolic variable $\psi^{\rho}(x)$ to represent the type of the variable x. The superscript indicates this is a variable for *type*. We support equivalence and non-equivalence checking, and sub-typing on the values of these variables. We encode the subtype relation as a partial order between types, which is supported by the solver.

Explicitly Reasoning about Attributes. We introduce symbolic

Table 1: Tracing Rules for Typical Statements

Rule	Statement	Execution States	Events(eid and ℓ omitted)

(class-def)	`class C(B):`
	`  def m(self, x):`
	`  self.x = x`
		$\sigma \rightarrow \sigma[C \mapsto o_1 = newobject(),$
		$o_1.m \mapsto o_2 = newobject()]$
		$\rho \rightarrow \rho[C \mapsto ClassType,$
		$o_1.m \mapsto MethodType]$
		$\pi \rightarrow \pi[o_1 \mapsto \mathcal{F}_{baset}(ClassType) \cup \{\text{“}m\text{”}\},$
		$o_2 \mapsto \mathcal{F}_{baset}(MethodType)]$
		$T \rightarrow T \cup \{CType \preccurlyeq BType\}$
			$e_1 : \langle C{=}o_1, (o_1, \sigma[C], \pi[o_1])\rangle$
			$e_2 : \langle C.m{=}o_2, (o_2, \sigma[o_1.m], \pi[o_2])\rangle$
			$\Gamma \rightarrow \Gamma; e_1; e_2$

(func-def)	`def f(x):` `  return x`	$\sigma \rightarrow \sigma[f \mapsto o_1 = newObject()]$ $\rho \rightarrow \rho[f \mapsto FuncType]$ $\pi \rightarrow \pi[o_1 \mapsto \mathcal{F}_{baset}(FuncType)]$	$\Gamma \rightarrow \Gamma; \langle f{=}o_1, (o_1, \sigma[f], \pi[o_1])\rangle$
(list-alloc)	`y=[1,2]`	$\sigma \rightarrow \sigma[y \mapsto o_1, o_1\_0 \mapsto 1, o_1\_1 \mapsto 2]$ $\rho \rightarrow \rho[y \mapsto list, o_1\_0 \mapsto int, o_1\_1 \mapsto int]$ $\pi \rightarrow \pi[o_1 \mapsto \mathcal{F}_{baset}(list)]$	$e_1 : \langle y{=}o_1, (o_1, list, \mathcal{F}_{baset}(list))\rangle$ $e_2 : \langle y\_0{=}1, (1, int, \mathcal{F}_{baset}(int))\rangle$ $e_3 : \langle y\_1{=}2, (2, int, \mathcal{F}_{baset}(int))\rangle$ $\Gamma \rightarrow \Gamma; e_1; e_2; e_3$
(class-instant)	`y=C() # C is a class`	$\sigma \rightarrow \sigma[y \mapsto (o_1 = newObject())]$ $\rho \rightarrow \rho[y \mapsto CType]$ $\pi \rightarrow \pi[o_1 \mapsto \pi[\sigma[C]]$	$\Gamma \rightarrow \Gamma; \langle y = o_1, (o_1, CType, \pi(C))\rangle$
(simple-assign)	`y=x`	$\sigma \rightarrow \sigma[y \mapsto \sigma(x)]$ $\rho \rightarrow \rho[y \mapsto \rho(x)]$	$\Gamma \rightarrow \Gamma; \langle y{=}x, (\sigma(y), \rho(y), \pi(\sigma(y)))\rangle$
(bop-assign)	`z=x bop y`	$\sigma \rightarrow \sigma[z \mapsto \sigma[x]\ bop\ \sigma[y]]$ $\rho \rightarrow \rho[z \mapsto \mathcal{F}_{btype}(\rho[x], \rho[y], bop)]$ $\pi \rightarrow \pi[\sigma(z) \mapsto F_{baset}(\rho[z])]$	$\Gamma \rightarrow \Gamma; \langle z{=}x\ bop\ y, (\sigma[z], \rho[z], \pi[\sigma[z]])\rangle$
(attr-read)	`y=x.f`|`y=getattr(x, 'f')`	$\sigma \rightarrow \sigma[y \mapsto \sigma[\sigma[x].f]]$ $\rho \rightarrow \rho[y \mapsto \rho[\sigma[x].f]]$	$\Gamma \rightarrow \Gamma; \langle y{=}x.f, (\sigma[y], \rho[y], \pi[\sigma[y]])\rangle$
(attr-write)	`y.f=x`|`setattr(y,'f',x)`	$\sigma \rightarrow \sigma[\sigma[y].f \mapsto \sigma[x]]$ $\rho \rightarrow \rho[\sigma[y].f \mapsto \rho[x]]$ $\pi \rightarrow \pi[\sigma[y] \mapsto \pi[\sigma[y]] \cup \{\text{“}f\text{”}\}]$	$\Gamma \rightarrow \Gamma; \langle y.f{=}x, (\sigma[x], \rho[x], \pi[x])\rangle$
(for-loop)	`for x in y...`	$i \leftarrow \mathcal{F}_{index}(\sigma[y])$ $\sigma \rightarrow \sigma[x \mapsto \sigma[\sigma[y]\_i]]$ $\rho \rightarrow \rho[x \mapsto \rho[\sigma[y]\_i]]$	$\Gamma \rightarrow \Gamma; \langle x{=}y\_i, (\sigma[x], \rho[x], \pi[\sigma[x]])\rangle$
(if-cmp)	`if(x relop y)...`	$r \leftarrow (\sigma[x]\ relop\ \sigma[y])$	$\Gamma \rightarrow \Gamma; \langle relop(x, y), (r, bool, \mathcal{F}_{baset}(bool))\rangle$
(if-isinst)	`if(isinstance(x, list))...`	$r \leftarrow (\rho[x] \preccurlyeq list)$	$\Gamma \rightarrow \Gamma; \langle isinst(x, list), (r, bool, \mathcal{F}_{baset}(bool))\rangle$
(if-hasattr)	`if(hasattr(x, 'f'))...`	$r \leftarrow (\text{“}f\text{”} \in \pi[\sigma[x]])$	$\Gamma \rightarrow \Gamma; \langle hasattr(x, \text{“}f\text{”}), (r, bool, \mathcal{F}_{baset}(bool))\rangle$
(call)	`y=f(a)`	$\sigma \rightarrow \sigma[b \mapsto \sigma[a]]$ $\rho \rightarrow \rho[b \mapsto \rho[a]]$ (b *is the formal parameter*)	$e_1 : \langle b = a,\ (\sigma[a], \rho[a], \pi[\sigma[a]])\rangle$ $e_2 : \langle y = f(a), (\sigma[\sigma[a].f],$ $FuncType, \mathcal{F}_{baset}(FuncType))\rangle$ $\Gamma \rightarrow \Gamma; e_1; e_2$

variable $\psi^{\pi}(obj)$ to represent the attribute-set of an object obj. Note that the attribute sets are associated with objects instead of variables following the Python memory model. In particular, if x and y are aliases through the assignment $y = x$, then the statement $x.f = ...$ adds the attribute "f" to the object that both x and y can reference. We represent the attribute set as a set in the constraint system and support the membership operation.

Encoding Rules. Before encoding, we first transform the trace into its *single static assignment* (SSA) form so that every variable is defined exactly once. We use x^i to denote the definition instance of x at trace entry i. We also extend the trace with statements from unexecuted branches (see Section 5.2). As such, ϕ-functions are also introduced to integrate definitions from two branches of a conditional statement at the join point.

Table 2 shows the encoding rules for the typical events. Rules (1)-(4) represent the encodings for the basic events, (5) represents the encodings for ϕ-functions, and rules (6)-(8) denote the encodings for the heap related events.

Rule (1) specifies the encoding for the assignment with a constant. The symbolic value variable $\psi^{\sigma}_{\tau}(z^i)$ is asserted to the value v. The symbolic type variable is asserted to type τ, and the object created to represent the value of z^i, which is acquired by a function $obj(z^i)$, is asserted to have the attribute set A.

Rule (2) specifies the encoding for a simple assignment, which makes z^i and x^j aliases. In the encoding, since we introduce a set of symbolic variables of different types to denote the value of a program variable of various types, for each type τ, we assert the equivalence of the corresponding (symbolic) value of z^i and x^j. This rule does not affect the attribute set, which is associated with the object instead of the variable.

Rule (3) encodes the addition operation. The basic idea is that for each type that has the addition operation, we encode its behavior. For simplicity, we assume the solver only supports additions for int and string types. As such, the first two clauses assert integer addition and string concatenation, respectively. The third one encodes additions that are not supported by the solver such that we cannot reason about their value variations. We assert the value, type, attribute set of z^i to the observed ones. More importantly, we also need to assert the operands x^j and y^k must hold their observed values to ensure soundness.

Rule (4) encodes a type check predicate. It simply asserts the type of the variable is a subclass of the intended type τ.

Rule (5) specifies the encoding for the ϕ function that joins the updates of a variable in the two branches of a predicate: z^i is equal to z^t if the predicate e_k holds, and is equal to z^f otherwise, where z^t and z^f are the most recent versions of z inside the branches. In the encoding, all the symbolic value and type variables are updated according to the encoding of e_k, denoted as $\Phi(e_k)$. The encoding of object attribute sets is also handled path-sensitively, similar to ϕ functions for variables. In other words, we also join the attribute sets of an object in a path sensitive fashion at a merge point of branches. Details are elided.

Rule (6) encodes an object attribute read. The encoding relies on a pre-computed points-to set $pts(x^j)$ representing the set of objects that may be pointed to by x^j. We use the standard Anderson [10] algorithm to compute the may points-to set on the trace (extended with the un-executed branches). Intuitively, our extended trace can be considered as a program with loop unrolled and dynamic features uncovered. In the encoding, for the object o in the points-to set that x^j actually points-to, we assert the equivalence of the values of $o.f^k$ and y^i and the equivalence of their types. Note that the superscript k denotes the most recent definition point of $o.f$. Function $type(o)$ represents the type of object o.

Rule (7) encodes an object attribute write. For the object o pointed-to by y^j, we assert the equivalence of the values of $o.f^i$ and x^k and the equivalence of their types. Besides, we assert the new attribute set is the union of the old attribute set and the "f". If the object is not pointed to by y^j, we

Table 2: Encoding Rules. The events are in the SSA form with the superscript of a variable x representing the id of the event where x is most recently defined. $obj(x)$ represents the object created for variable x. $\psi^{\sigma}(x)$, $\psi^{\rho}(x)$, and $\psi^{\pi}(x)$ represent the symbolic variables denoting value, type, and attribute set for a program variable/value x. Each type τ means each possible type of x. $type(o)$ denotes the type of object o.

SSA Formed Event(e_i)	Encoded Symbolic Constraints
(1) $\langle z^i = v, (v, \tau, A)\rangle$	$\psi^{\sigma}_{\tau}(z^i) = v \wedge \psi^{\rho}(z^i) = \tau \wedge \psi^{\pi}(obj(z^i)) = A$
(2) $\langle z^i = x^j, ...\rangle$	$(\bigwedge_{\text{each type } \tau} \psi^{\sigma}_{\tau}(z^i) = \psi^{\sigma}_{\tau}(x^j)) \wedge \psi^{\rho}(z^i) = \psi^{\rho}(x^j)$
(3) $\langle z^i = x^j + y^k, (v, \tau, A)\rangle$	$\texttt{if } (\psi^{\rho}(x^j) = int \wedge \psi^{\rho}(y^k) = int) \Rightarrow (\psi^{\sigma}_{int}(z^i) = \psi^{\sigma}_{int}(x^j) + \psi^{\sigma}_{int}(y^k) \wedge \psi^{\rho}(z^i) = int \wedge \psi^{\pi}(obj(z^i)) = A_{int})$ $\texttt{elif } (\psi^{\rho}(x^j) = str \wedge \psi^{\rho}(y^k) = str) \Rightarrow (\psi^{\sigma}_{str}(z^i) = concat(\psi^{\sigma}_{str}(x^j), \psi^{\sigma}_{str}(y^k)) \wedge \psi^{\rho}(z^i) = str \wedge \psi^{\pi}(obj(z^i)) = A_{str})$ $\texttt{else } (\psi^{\sigma}_{\tau}(z^i) = v \wedge \psi^{\rho}(z^i) = \tau \wedge \psi^{\pi}(obj(z^i)) = A) \wedge \psi^{\sigma}_{type(obj(x^j))}(x^j) = obj(x^j) \wedge \psi^{\sigma}_{type(obj(y^k))}(y^k) = obj(y^k)$
(4) $\langle if\ isinst(x^j, \tau), ...\rangle$	$\psi^{\rho}(x^j) \preccurlyeq \tau$
(5) $z^i = \phi(e_k, z^t, z^f)$	$\Phi(e_k) \Rightarrow ((\bigwedge_{\text{each type } \tau} \psi^{\sigma}_{\tau}(z^i) = \psi^{\sigma}_{\tau}(z^t)) \wedge \psi^{\rho}(z^i) = \psi^{\rho}(z^t)) \wedge$ $\neg\Phi(e_k) \Rightarrow ((\bigwedge_{\text{each type } \tau} \psi^{\sigma}_{\tau}(z^i) = \psi^{\sigma}_{\tau}(z^f)) \wedge \psi^{\rho}(z^i) = \psi^{\rho}(z^f))$
(6) $\langle y^i = x^j.f, ...\rangle$	$\bigwedge_{o \in pts(x^j)} (\psi^{\sigma}_{type(o)}(x^j) = o) \Rightarrow ((\bigwedge_{\text{each type } \tau} \psi^{\sigma}_{\tau}(y^i) = \psi^{\sigma}_{\tau}(o.f^k)) \wedge \psi^{\rho}(y^i) = \psi^{\rho}(o.f^k))$
(7) $\langle y^j.f = x^k, ...\rangle$	$(\psi^{\sigma}_{type(o)}(y^j) = o) \Rightarrow ((\bigwedge_{\text{each type } \tau} \psi^{\sigma}_{\tau}(o.f^i) = \psi^{\sigma}_{\tau}(x^k)) \wedge \psi^{\rho}(o.f^i) = \psi^{\rho}(x^k) \wedge \psi^{\pi}(o)' = \psi^{\pi}(o) \cup \{\text{“}f\text{”}\}) \wedge$ $\bigwedge_{o \in pts(y^j)} (\psi^{\sigma}_{type(o)}(y^j) \neq o) \Rightarrow ((\bigwedge_{\text{each type } \tau} \psi^{\sigma}_{\tau}(o.f^i) = \psi^{\sigma}_{\tau}(o.f^p)) \wedge \psi^{\rho}(o.f^i) = \psi^{\rho}(o.f^p) \wedge \psi^{\pi}(o)' = \psi^{\pi}(o))$
(8) $\langle if\ hasattr(x^j, \text{“}f\text{”}), ...\rangle$	$\bigvee_{o \in pts(x^j)} (\psi^{\sigma}_{type(o)}(x^j) = o) \wedge (\text{“}f\text{”} \in \psi^{\pi}(o))$

assert the attribute value and type, and the attribute-set of the base object stay the same. Note that the superscript i and p denote the current and the most recent definition points of the object attribute, respectively. We use $\psi^{\pi}(o)'$ to denote a new defined version of the symbolic attribute set of o.

Rule (8) encodes an attribute check. For any object o in the points-to set of x^j, if the object referred to by x^j equals o, the attribute “f” must be in the attribute set of o.

Handling Solver Unsupported Types. Python supports many more types compared to those supported by the solver. In our encoding rules, we do not specify the details of handling solver unsupported types for space reason. In fact, we treat the values of solver unsupported types as objects denoted by IDs. We only reason about equivalence and non-equivalence of object IDs. This is consistent with existing work of predictive analysis [8, 7].

5.2 Unexecuted Path Relaxation

To allow reasoning about possible path changes induced by input variations, we need to include the unexecuted branches to our constraint system. The basic idea is to enhance the observed trace to aggressively include statements in unexecuted branches as long as those statements can be fully resolved by static analysis. We preclude the unresolved branches by asserting the branch outcome as the observed one. This ensures the soundness of our analysis, although it limits the coverage. Next, we explain how we determine if an unexecuted statement can be resolved.

Assignments. Constant assignments $x = v$ can always be resolved and encoded. For a simple asssignment $y = x$, our analysis first looks for the most recent definition of x. If it may be defined in an unresolved branch, which was hence not encoded, we consider the variable x unresolved. As a result, the assignment and the whole branch are also unresolved. Assignments of binary operations like $z = x + y$ are handled similarly.

Conditionals and Loops. For a conditional statement `if` $x > y$ s_1 `else` s_2 inside an unexecuted branch, we first try to resolve x and y and then s_1 and s_2. If any of these cannot be resolved, the conditional is not resolved, except the following case. If x and y can be resolved, and one of the branches can be resolved, say s_1. We encode s_1 and assert the branch outcome to be true. For a loop statement (in an unexecuted branch), we require the loop bound can be statically resolved as constant.

Calls. For a function call in an unexecuted branch, we must ensure it is not a recursive call or an unmodeled external function. If this is satisfied and the callee can be resolved, we say the invocation can be resolved. If a call statement has multiple possible callees and only some of them can be resolved, we assert the call target falls into the set that can be resolved. We do not handle try-except statements. If we encounter any such statements in an unexecuted branch, the branch is considered unresolved.

Example. Consider the example in Fig. 6. The left column is the source code and the shaded statements are executed by the observed run. Note that lines 4 and 6-8 are not executed. The right column shows the intermediate SSA form of the trace after we extend it with the unexecuted branches. Note that we are able to resolve and add line 4 as y can be resolved. As a result, a ϕ statement is added at line 14. In contrast, the branch consisting of lines 6-8 cannot be resolved as we cannot statically resolve the loop bound y at line 7. As such, we assert the branch outcome of line 5 to be false.

5.3 Constraint Solving for Bug Detection

After the extended trace is fully encoded, we further add assertions for bug detection before the constraints are passed to the solver. While the user can come up various customized bug detection assertions. In our current system, we mainly focus on two kinds of assertions.

Subtype Assertion. Subtype assertion is used to detect type errors. More specifically, we assert if the type of a variable may fall out of an acceptable range. For example, suppose the acceptable types of variable x include *int* or *str*. We assert $\neg(\psi^{\rho}(x) \preccurlyeq int \vee \psi^{\rho}(x) \preccurlyeq str)$. For operations involving multiple variables, we check the type combinations of these variables. For example, for a binary operation $z = x + y$, the Python interpreter does not allow the addition of a string and a number. Therefore, we assert x is an integer and y is string or vice versa.

Attribute Assertion. Attribute assertion is used to check if a non-existent attribute may be read. More specifically, for an attribute read $y = x.f$, let points-to set $pts(x) =$

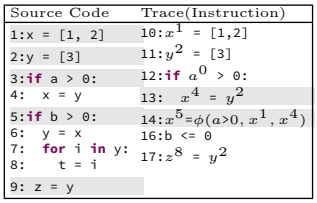

Figure 6: Unexecuted Path Relaxation.

$\{obj_1, ..., obj_n\}$, we check if there exists $obj_t \in pts(x)$ that satisfies: (1) there is a feasible path making $x = obj_t$ satisfiable, and (2) obj_t does not contain the attribute "f".

We send all the constraints to the solver. If the result is SAT, we find a bug and its triggering input.

6. SOUNDNESS GUARANTEE

Our technique is sound, meaning that it does not have false positives. The soundness guarantee can be reasoned as follows. Our handling of loop is sound since we assert the loop is unrolled for the same times as in the observed run (therefore prohibiting certain input variations) and require the iteration count of the loop in the unexecuted branch to be resolved as constant at compile time. Our handling of the types (e.g., the user specified types) that are not supported by the solver is sound. We limit the behaviors of such objects to only the equivalence and nonequivalence check, and therefore, declaring them as integers (e.g., object ids) in the solver is sufficient. In terms of the external functions, we manually model the commonly used functions, and for the rest, we automatically detect those with side effects and conservatively under-approximate their behaviors by fixing their input/output pairs. We assume that the manually created models are correct and the functions with side effects always change the values of the involved variables. Under the assumption, our handling of external functions is sound. Besides, our handling of the heap is sound as we model all heap accesses and updates. We allow both the base reference and the attribute set of an attribute access to change. Note that our reasoning of heap is with respect to the scope of the trace and the extended branches, rather than the whole program. The quality of the results depends on the test suite, just like all the predictive analyses [4, 5, 6, 7].

7. EVALUATION

We have implemented a prototype in Python. The system mainly consists of a tracing infrastructure and a constraint encoding and solving component. As mentioned in earlier sections, Python programs tend to use a lot of external functions and classes implemented in other languages. We manually modeled a set of commonly used external functions by rewriting them in Python. We use Z3 as the SMT solver. To improve scalability, we also implement a dynamic slicer to prune unrelated trace entries for a given assertion. We make our prototype along with the evaluation environment publically available at [31].

Our evaluation aims to address the following questions:

RQ1: How effective is our approach in detecting potential bugs for real world Python projects?

RQ2: How efficient is our approach? Can it scale to real world programs and executions?

RQ3: How applicable is it to use the test cases provided in the projects as the observed runs? Can our approach detect more bugs by increasing test coverage?

To answer the above research questions, we apply our approach to a set of real world Python projects, as shown in Table 3 (columns 1-2). These benchmarks are mainly from GitHub. All of them are highly ranked in popularity and widely used in practice. To select these benchmarks, we went through the ranking of popularity and bug reports, and selected those that have the highest ranking and at least one reproducable bug report. We also considered diversity during selection. Observe that some of them are large. The largest one has over 84K source lines of Python code. The sum of the source code lines of these projects is about 164K.

These projects fall in the following application domains, which are representative for Python programs:

`Requests` and `urllib3` - very popular HTTP libraries.

`Fabric` and `salt` - well-known remote deployment and system administration applications.

`Flask`, `bottle` and `web2py` - widely used Web application development frameworks.

`Tweepy`, `boto` and `praw` - well known web service APIs.

`Bs4` - a very popular HTML and XML parsing library.

Our experiments were conducted on a machine with an Intel i5-4210U, 4GB RAM, Ubuntu 14.04, and Python 2.7.4.

7.1 Experimental Results

Table 3 shows the experimental results. For each program, we run the (passing) test cases provided as part of the project. We then encode these traces together with unexecuted branches to find bugs. Many of the test cases do not lead to the identification of bugs. In Table 3, we only report the statistics for those test runs that allow us to identify bugs.

Trace and Constraints. Column 3 (`Mod`) presents the number of modules executed, which ranges from 2 to 36. Note that we have excluded some uninteresting modules such as those related to logging and multiprocessing. Columns 4-7 report the number of instructions that fall into some popular categories, namely, attribute reads (`AttrR`), attribute writes (`AttrW`), predicates (`Pred`) and calls. Column 8 presents the total number of events in the trace. Observe that many of these executions are long. The longest execution has over 70K events.

Column 9 (`AP`) presents the number of events encoded when reaching the buggy assertion point. Recall that our system adds assertions in order to find bugs (Section 5.3). Column 10 (`ExtE`) presents the number of unexecuted statements that are added to the encoding in column 9. Observe that our technique was able to resolve a lot of unexecuted branches, allowing us to explore a large number of neighboring executions of the observed run. Column 11 (`Cons`) presents the total number of constraints when reaching the assertion of the violation, which ranges from 41 to over 135K.

Detected Defects (RQ1). Our system is able to detect a set of bugs for these programs. The reported bugs are all true positives. Column 12 presents the bug issue ids if available. The issue ids prefixed with * are the new bugs that have not been reported before. We reported these new defects to the developers through the bug tracker of `Github` or `launchpad`. Most of them have been confirmed and even fixed. Id "`*s22547`" means that the root cause of the bug is very similar to that of issue "`22547`". There are three known bugs that we cannot find the issue numbers. Instead, we provide the commit records. Column 13 presents the number of the detected defects. Note that our approach is effective, identifying 46 defects with 30 known and 16 new.

Table 3: Statistics of Experimental Results

(a)Benchmarks		(b)Trace						(c)Constraints			(d)Defects		(e)Time(sec)		
Name	SLOC	Mod	AttrR	AttrW	Pred	Calls	Events	AP	ExtE	Cons.	Issue#	Count	Tracing	Solving	Total
requests	9934	36	1257	2506	1022	1331	18558	6468	1223	11461	1462	8+4	6.89	84.74	91.63
		8	126	37	72	107	982	71	0	144	2077		0.62	0.83	1.45
		35	1680	3566	1537	1738	26436	8107	1445	14378	1711		9.31	96.08	105.39
		31	1159	2360	968	1230	17225	11980	2494	20965	1082		5.12	121.52	126.64
		30	1350	10097	1108	1406	42092	7309	1318	12886	1324		8.3	78.96	87.26
		33	1285	10192	1078	1320	41707	10656	2201	18669	1019		8.03	106.71	114.74
		34	975	2449	830	1155	15943	7852	1460	13763	2613		11.9	98	109.9
		10	86	223	101	160	1801	71	5	139	2267		1.27	1.87	3.14
		11	140	258	144	196	1257	2819	651	4887	*2527		0.97	29.1	30.07
		12	266	353	232	330	3708	4467	878	7910	*2552		1.59	42.03	43.62
		29	902	1942	782	1059	13623	6734	1261	11781	*2638		6.31	84.4	90.71
		35	982	2348	834	1161	15666	2824	534	5093	*2639		6.64	40.4	47.04
urllib3	4798	7	41	38	20	43	406	168	15	324	355	2	0.31	1.52	1.83
		10	347	8695	310	387	28801	505	69	836	143		4.29	6.88	11.17
fabric	2995	10	188	143	179	169	2318	289	33	609	1204	6+1	1.61	4.2	5.81
		6	31	25	13	32	302	243	17	482	610		0.24	3.32	3.56
		2	6	0	4	7	57	30	3	55	1096		0.06	0.69	0.75
		4	6	0	5	7	57	30	3	55	1096		0.07	0.63	0.7
		3	20	14	14	11	168	111	18	195	1191		0.28	1.24	1.52
		2	9	0	8	6	71	57	15	100	1096		0.08	0.83	0.91
		3	21	14	15	16	171	117	23	196	*1303		0.15	1.64	1.79
salt	84789	13	1264	330	594	863	7341	12667	5368	17978	5959	6+7	12.26	49.85	62.11
		13	2438	2806	1970	2058	30099	10930	2436	18988	11873		13.3	106.91	120.21
		14	1298	356	630	903	7736	12869	5432	18327	12651		7.15	89.9	97.05
		5	19	11	11	15	160	23	2	41	6817		0.49	0.35	0.84
		19	2296	694	1320	1533	15385	13122	5436	18837	12453		9.76	90.1	99.86
		4	9	25	11	27	198	227	29	398	8635		0.4	2.42	2.82
		14	1303	358	632	906	6696	12781	5290	18184	*22547		6.99	88.66	95.65
		14	1341	306	644	942	7728	12515	5359	17689	*s22547		9.55	109.24	118.79
		15	1344	304	648	927	7686	12635	5414	17834	*24820		11.36	122.27	133.63
		15	1817	904	1148	1678	15291	17563	6540	26468	*25006		10.47	129.33	139.8
		15	1818	904	1149	1678	15295	14110	5566	20702	*s25006		10.99	106.3	117.29
		15	1817	904	1149	1678	15290	17571	6540	26486	*s25006		9.64	128.11	137.75
		19	1591	599	837	1433	11678	15398	6028	22692	*s25006		11.11	156.91	168.02
flask	2645	9	2206	1014	1650	941	15975	21497	5696	33615	857	2	13.01	351.72	364.73
		25	10142	2456	4320	2333	70361	76215	8210	135908	237		67.23	1549.81	1617.04
bottle	3003	4	150	104	99	156	1625	1468	214	2680	223	1	0.71	18.03	18.74
web2py		2	16	13	8	14	125	48	7	80	11/01/12	1+1	1.28	1.32	2.6
		2	5	6	11	7	80	68	17	101	*968		3.97	1.29	5.26
tweepy	2394	19	68	198	69	112	1316	1483	218	2748	12/22/13	1+1	1.25	26.9	28.15
		7	9695	1840	3707	5284	61571	67406	5899	121255	*642		36.25	1165.21	1201.46
boto	47904	18	265	265	158	45	2619	2766	535	4840	03/19/12	1	3.07	70.19	73.26
praw	3034	4	87	152	86	68	1324	19	4	38	87	1	7.22	0.4	7.62
bs4	2718	7	41	42	77	79	771	44	10	79	1307471	1+2	0.53	0.68	1.21
		5	83	50	101	83	921	171	47	254	*1463984		0.64	3.42	4.06
		5	66	42	81	75	783	160	44	244	*1463984		1.34	3.17	4.51
Total	164214	-									30+16=46		-		

Analysis Time (RQ2). Columns 14-16 present the tracing time, solving time, and total time, respectively. The solving time is the sum of the many queries to the solver regarding the bug identification assertions. From the results, we argue that the cost of our approach is reasonable. It detects most of these defects in a few seconds to a few minutes. The most time-consuming case is issue 237 of *flask*, which took about 27 minutes to process. Further inspection shows that it is due to the long execution trace.

Fault Detection Capability versus Test Coverage (RQ3). The detection capability of our approach largely depends on the observed runs. In our experiment, we mainly choose the test cases along with the projects as the seed runs. The reason is that test-driven development is pervasive in programming with dynamic languages. As such, there are often many test cases along with Python projects.

We randomly choose 100 passing test cases from the test-suites of two benchmarks, `requests` and `salt`, which have the largest number of defects. We then study how the accumulative number of detected bugs grows with the test coverage. Fig. 7(a) shows the result. Note that here we measure statement coverage. The result reveals a close-to-linear relation between the two. We hence argue that our approach can detect more defects with more test cases to achieve better coverage. In some sense, these test cases drive the execution to interesting places and our technique reasons about bugs in the neighboring executions. Fig. 7(b) shows the accumulative analysis time of our system. Observe that the growth is steady and the total time is reasonable.

7.2 Defect Study

To better understand the common patterns of defects in Python programs, we further examine these reported bugs and classify them into three categories as shown in Table 4.

Attribute Errors. 23(50%) of the detected defects cause *AttributeError* crashes, which mean that the execution accesses a nonexistent attribute. In Python programs, it is possible that an attribute is added to the object in one path but not in another path. Developers may forget the latter path and access the attribute which does not exist.

More specifically, there are 6 such defects in `requests` with 2 new defects. Take the new issue 2527 as an example. `PreparedRequest.copy()` invokes the `copy()` method of object `self._cookies`, which is passed from some high level API (e.g., the instantiation method of class `Request`). Unfortunately, some of these APIs may not add the `copy()` method to the attribute set. There are 2 attribute defects in `fabric`. For instance, in issue 610, the `key_filename()` function of the

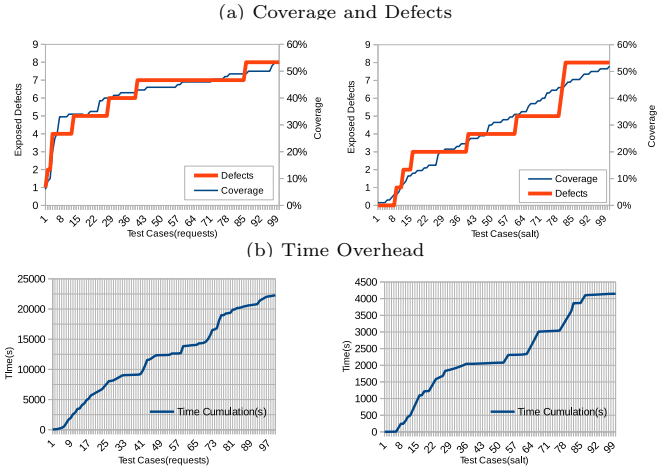

Figure 7: Bug Detecting Capability along with Coverage

`network.py` module checks if `env.key_filename` is an instance of `list` type. If not, it will wrap `env.key_filename` into a `list`. Later on, it reads the "`startswith`" attribute of each item in the `list`. Our approach finds that the program will crash with an *AttributeError* raised when `env.key_filename` is initialized as another iterable type object (e.g., a `set`). This is because the program wraps the set object into a list and reads the "`startswith`" attribute of the set object, which does not exist.

Type Errors. 3 (6.52%) of the detected defects cause *TypeError* crashes, which occur when a parameter of an invalid type is passed to a built-in function or when a binary operation has unmatched operands. The type error bug in `requests` is shown in Fig. 1 and discussed. The bug in `salt` occurs if the parameter passed to `time.sleep()` is not of a number type (e.g., int and float).

Unicode Errors. 20 (43.48%) of the reported defects cause *UnicodeEncodeError* or *UnicodeDecodeError* crashes. Unicode errors are very common in Python applications, especially in Web related applications. The Python 2.x interpreter implicitly casts a byte string to a unicode string during certain operations (e.g., concatenation between a byte string and a unicode string). However, such casting crashes if the input string contains non-ascii characters. Specifically, we have found 5 such bugs in `requests`, and 2 of them are new bugs. In issue 1082, `requests` provides a set of HTTP APIs (e.g., `get`/`put`) that support values of different types for the parameters `headers` and `data`. Both parameters allow unicode and byte strings. The program crashes when `headers` contains a unicode string (e.g., `headers={u"Content-Type": ...}`) and `data` contains a byte string with non-ascii characters. This is because an internal method `_send_output()` of module `httplib.py` concatenates them, triggering the implicit cast and causing the crash. We have also found unicode bugs in `salt`, all of which are not reported before.

8. RELATED WORK

Our approach belongs to the category of predictive analysis [4, 5, 8]. Existing predictive analyses were originally used in concurrency program analysis, such as race detection [4, 5, 6, 7] and atomicity violation detection [8]. They focus on changing the schedules to explore different executions. Differently, our approach is the first to change inputs to explore different executions. Besides, our approach allows the exploration of the unexecuted branches, while existing approaches do not. Our handling of the Python (dynamic) features, such as typing and attribute sets, is also novel.

Table 4: Classifications of Detected Defects

Benchmarks	AttributeError		TypeError		UnicodeError	
	Known	New	Known	New	Known	New
requests	4	2	1	0	3	2
urllib3	2	0	0	0	0	0
fabric	2	0	1	0	3	1
salt	5	1	1	0	0	6
flask	1	0	0	0	1	0
bottle	1	0	0	0	0	0
web2py	0	0	0	0	1	1
tweepy	0	0	0	0	1	1
boto	1	0	0	0	0	0
praw	1	0	0	0	0	0
bs4	1	2	0	0	0	0
Total(Ratio)	23(50%)		3(6.52%)		20(43.48%)	

The symbolic execution [15] can be combined [16] with the evolutionary search, in which a fitness function is designed and used to drive the exploration of the input space. The design of the fitness function varies across different research problems and takes the great manual efforts. Researchers [17] further propose an algorithm to automatically generate the fitness functions. Besides, the symbolic execution can be optimized [18] by leveraging the read/write analysis to prune the methods that do not interact with the side effects of interest. However, all these approaches apply to traditional programs and cannot handle Python programs due to the lack of the symbolic reasoning of the dynamic language features.

Several tools have been developed for Python in bug detection [2, 3] and symbolic (concolic) execution [19, 21]. Bug detection tools are mainly based on static analysis and handle defects that do not involve dynamic features. Recent advances in Python symbolic (concolic) execution [19, 21] have shown good potentials. These techniques gradually explore paths within the search space, driven by a path exploration engine. In contrast, we use test cases to drive execution to interesting states and explore bugs in the neighborhood. Our state exploration is done inside the solver, not by a path exploration engine. As such, we need to address some new challenges such as encoding the value of a variable in various types, handling unexecuted paths, and modeling attribute set changes. Type inference for Python [27, 28, 29] has also been proposed. The work is mainly based on abstract interpretation, which may have difficulty handling dynamic features.

A set of analyses have been proposed for other dynamic languages such as Javascript [11, 12, 22, 24], Ruby [14, 13, 23] and PHP [30, 25, 26]. TypeDevil [11] detects type inconsistencies through checking if a variable or field may have multiple types in the same run. Jalangi [22] supports concolic testing for Javascript which enables detecting type inconsistencies. Rubydust [13] dynamically infers types to report errors at method boundaries. Rubyx [23] uses symbolic execution to analyze Ruby-on-Rails web applications for security vulnerabilities. Zheng et al. [26] proposed a path-sensitive static analysis of PHP web applications for security vulnerability detection.

9. CONCLUSION

In this paper, we present a Python predictive analysis. It has two steps to detect bugs in Python programs. The first step is to collect the execution trace of a passing run. The second is to encode the collected trace and some unexecuted branches into symbolic constraints. By solving the constraints, we can identify bugs as well as their triggering inputs. Our evaluation shows that our technique is able to detect 46 bugs with 16 unreported before. All of them are true positives.

10. REFERENCES

[1] IEEE Spectrum. http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages.

[2] PyChecker. http://sourceforge.net/projects/pychecker/.

[3] PyLint. http://www.pylint.org/.

[4] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C Flanagan. Sound predictive race detection in polynomial time. In *POPL*, 2012.

[5] J. Huang, P. O. Meredith, and G. Rosu. Maximal sound predictive race detection with control flow abstraction. In *PLDI*, 2014.

[6] M. Said, C. Wang, Z. Yang, and K. Sakallah. Generating data race witnesses by an smt-based analysis. In *NFM*, 2011.

[7] C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In *FM*, 2009.

[8] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In *FSE*, 2010.

[9] A. Farzan, P. Madhusudan, N. Razavi, and F. Sorrentino. Predicting null-pointer dereferences in concurrent programs. In *FSE*, 2012.

[10] L. O. Andersen. Program Analysis and Specialization for the C Programming Language. *PhD's thesis, UCPH*, 1994.

[11] M. Pradel, P. Schuh, and K. Sen. TypeDevil: Dynamic type inconsistency analysis for JavaScript. In *ICSE*, 2015.

[12] L. Gong, M. Pradel, M. Sridharan, and K. Sen. DLint: dynamically checking bad coding practices in JavaScript. In *ISSTA*, 2015.

[13] J. D. An, A. Chaudhuri, J. S. Foster, and M. Hicks. Dynamic inference of static types for Ruby. In *POPL*, 2011.

[14] J. D. An, A. Chaudhuri, and J. S. Foster. Static Typing for Ruby on Rails. In *ASE*, 2009.

[15] S. Anand, C. Pasareanu, and W. Visser. JPF–SE: A Symbolic Execution Extension to Java PathFinder. In *TACAS*, 2007.

[16] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *DSN*, 2009.

[17] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. Vos. Symbolic search-based testing. In *ASE*, 2011.

[18] E. F. Rizzi, M. B. Dwyer , S. Elbaum. Safely reducing the cost of unit level symbolic execution through read/write analysis, In *ACM SIGSOFT Software Engineering Notes*, 2014

[19] T. Ball and J. Daniel Deconstructing Dynamic Symbolic Execution In *Proceedings of the 2014 Marktoberdorf Summer School on Dependable Software Systems Engineering*, 2014

[20] DART: directed automated random testing In *PLDI*, 2005.

[21] S. Bucur, J. Kinder, and G. Candea. Prototyping Symbolic Execution Engines for Interpreted Languages. In *ASPLOS*, 2014.

[22] K. Sen, S. Kalasapur, T. Brutch, et al. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript In *ESEC/FSE*, 2013.

[23] A. Chaudhuri and J.S. Foster. Symbolic security analysis of ruby-on-rails web applications. In *CCS*, 2010.

[24] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of javascript web applications. In *ICSE*, 2011.

[25] E. Kneuss, P. Suter, and V. Kuncak. Runtime instrumentation for precise flow-sensitive type analysis. In *RV*, 2011.

[26] Y Zheng, X. Zhang. Path sensitive static analysis of web applications for remote code execution vulnerability detection In *ICSE*, 2013.

[27] M .Salib. Starkiller: A static type inferencer and compiler for python. In *Master's thesis, MIT*, 2004.

[28] A. Rigo and S. Pedroni. Pypy's approach to virtual machine construction. In *OOPSLA*, 2006.

[29] M. Gorbovitski, Y. A. Liu, S. D. Stoller, T. Rothamel, and K. T. Tekle. Alias analysis for optimization of dynamic languages. In *DLS*, 2010.

[30] H. Zhao, I. Proctor, M. Yang, X. Qi, M. Williams, et al. The hiphop compiler for php. In *OOPSLA*, 2012.

[31] Python Predictive Analysis https://sites.google.com/site/pypredictor/