Dynamic Concolication for Automatic Unit Testing of Python Libraries

Chi-Rui Chiang
National Chengchi University
Taipei, Taiwan
111356024@nccu.edu.tw

Chih-Duo Hong
National Chengchi University
Taipei, Taiwan
chihduo@nccu.edu.tw

Fang Yu
National Chengchi University
Taipei, Taiwan
yuf@nccu.edu.tw

***Abstract*—Concolic testing, which combines concrete testing and symbolic execution, has proven highly effective in detecting software vulnerabilities. While traditional unit testing relies on manually crafted test cases to verify specific program behaviors, concolic testing automatically explores execution paths by mixing concrete and symbolic inputs. This paper presents a novel methodology for bridging these approaches by automating unit test generation through dynamic function tracking. Our approach enhances traditional concolic testing by dynamically wrapping functions and external libraries for symbolic execution, addressing the critical challenge of premature symbolic variable downgrading caused by unsupported operations. By dynamically upgrading inputs to concolic objects during function calls and integrating fuzzing techniques, our method maintains symbolic coverage across program execution and handles cases where direct symbolic upgrade is infeasible. Experimental results across various Python libraries demonstrate significantly higher code coverage and vulnerability detection compared to traditional methods, with the ability to generate comprehensive test suites from minimal initial inputs.**

***Index Terms*—Python, Unit Test, Concolic Testing, Runtime Analysis.**

1. Introduction

As software systems grow increasingly complex and ubiquitous, ensuring their reliability and security has become a critical challenge in modern computing. To address this challenge, the software industry has developed sophisticated automated testing techniques that can generate test cases, execute them, and analyze results without manual intervention. These techniques include fundamental approaches such as unit testing [1], fuzzing [2], symbolic execution [3], and concolic testing [4], each serving distinct yet complementary roles in comprehensive software validation. Among these methods, concolic testing (a portmanteau of “concrete” and “symbolic”) has emerged as a particularly effective approach for identifying software vulnerabilities. It achieves this by synthesizing two powerful testing paradigms: concrete testing, which evaluates system behavior using specific input data, and symbolic execution, which treats inputs as variables to generate constraint conditions. By alternating between these approaches, concolic testing systematically explores diverse system behaviors. This hybrid methodology effectively bridges the gap between black-box and white-box testing strategies, harnessing the advantages of both approaches [5].

However, one significant limitation of this approach involves the handling of unsupported operations and external functions. When the concolic testing algorithm encounters these constructs, it is forced to downgrade symbolic variables to their concrete values, thereby losing the symbolic information that is crucial for comprehensive path exploration [6]–[8]. The root cause of this limitation is tied to the algorithm’s reliance on an underlying SMT solver (like Z3 [9]), which only supports a subset of Python operations expressible in specific theories. Unsupported operations, such as bitwise operations or functions returning complex types like tuples or lists, cannot be easily expressed within these theories. Additionally, the symbolic representations are often inadequate for functions that behave like black-box operations, such as cryptographic hashes or system-level APIs, which are inherently non-symbolic [6]. This downgrading leads to gaps in test coverage and reduces the effectiveness of the concolic testing process.

To illustrate this issue, consider the following code snippet:

```python
import hashlib

def check_auth(user: str, pwd: str) -> bool:
  """
  Check if the user's hashed password matches
  the stored hash.
  """
  pwd_hash = hashlib.md5(pwd.encode()).hexdigest()
  return pwd_hash == stored_hash.get(user)
```

This code executes an external function hashlib.md5, which computes a cryptographic hash. This operation is inherently non-symbolic and behaves like a black box. The concolic engine cannot easily express the hashing operation symbolically, causing the symbolic input password to downgrade to its concrete value during testing. This loss of symbolic information prevents exploration of paths where different passwords might produce different hash values.

In this work, we propose to mitigate the symbolic downgrading problem by upgrading the input values of unsupported operations and external functions through separate concolic tests, thereby preserving the symbolic exploration capabilities. More specifically, we extend PyCT [7], a powerful Python concolic testing tool, by integrating a technique named Dynamic Function Tracking (DFT). DFT dynamically tracks function and function calls during execution and wraps these functions to keep inputs symbolic wherever possible. Existing concolic execution tools like CrossHair [8] are more static and tend to analyze functions symbolically only if these functions can be easily in-lined or if they have contracts. For complex functions nested inside other functions or classes, these tools often simply fall back to concrete execution. DFT, on the other hand, dynamically profiles and wraps functions. It monitors the execution of all functions at runtime and uses this profiling to decide whether the function can be symbolically executed or requires a different testing method. Whenever possible, DFT dynamically wrapped functions by converting concrete inputs into symbolic variables to enable more thorough testing. If symbolic execution is not feasible, it falls back to fuzzing, which generates random or mutated inputs to mitigate the loss of test coverage. Note that, while we have realized this technique on PyCT, it is not specific to PyCT and can be applied to any concolic engines facing similar challenges.

We evaluate the efficacy of our DFT approach by comparing it directly against the standard unit test framework of Python across a diverse collection of benchmarks. The comparison focuses on several key metrics, including code coverage, error detection, and dependency analysis. By dynamically tracking function calls and maintaining symbolic representations, DFT extends the capabilities of traditional unit testing. The experimental results reveal that DFT achieves significantly higher code coverage and identifies more errors compared to the baseline unit test approach, especially in testing libraries like Transformers, Jsonschema, and Numpydoc. Additionally, DFT demonstrates its strength in scenarios where the availability of predefined unit tests is limited, such as the CNN inference experiment, by dynamically generating inputs that reveal hidden errors in both the target package and its dependencies. Our evaluation also examines the integration of fuzzing with DFT. The results show that while this integration may extend testing coverage, the benefits come at the cost of increased execution time. These findings validate DFT's efficacy in enhancing automated testing for Python projects, especially in environments where traditional unit tests fall short.

Organization. This paper is organized as follows: Section 2 provides an overview of related work. Section 3 describes our framework and key contributions. Section 4 presents experimental evaluation across multiple Python libraries. Section 5 discusses the potential limitations of our approach, and Section 6 concludes the paper with directions for future work.

2. Related Work

Concolic testing has emerged as a powerful technique to overcome the inherent limitations of symbolic execution, such as path explosion and difficulties in solving non-linear constraints [5]. Initially, symbolic execution was introduced to verify software correctness by testing whether certain properties could be violated [10]. While it has been effective in tasks such as automated test case generation [11], program invariants inference [12], and vulnerability detection [13], purely symbolic execution struggles with the complexities of real-world programs, especially when encountering complex constraints or interacting with external APIs. Concolic testing addresses these challenges by executing programs with concrete inputs while collecting symbolic path constraints to explore alternative execution paths systematically [5]. By leveraging both symbolic and concrete execution, concolic testing can more efficiently explore a program's state space, avoiding the path explosion problem commonly seen in pure symbolic approaches. Successful applications of symbolic and concolic testing can be seen in tools like KLEE [14], SPF [15], and Triton [16].

Although symbolic and concolic execution engines have been widely adopted for languages like C, Java, and LLVM, their support for Python remains limited due to the language's dynamic typing and flexible data structures. In recent years, several concolic testing tools have been developed specifically for Python, including PyCT [7], CrossHair [8], PySym [17], and PEF [18]. These tools are built upon the pioneering architecture of PyExZ3 [19], which performs path exploration by substituting native data types with proxy objects and extracting symbolic constraints during execution. Nevertheless, such tools often encounter limitations when handling unsupported operations and external function calls, leading to symbolic downgrading where variables revert to their concrete values. While implemented on PyCT, our proposal to mitigate downgrading can in principle be integrated into all tools based on the PyExZ3 architecture.

In this work, we have further enhanced our dynamic testing techniques with fuzzing [20], [21] when downgrading is unavoidable. Many existing concolic testing tools incorporate fuzzing capabilities, including KLEE [14], Distiller [22], QSYM [23], and CrossHair [8]. However, only CrossHair and our work specifically target Python programs through designs that accommodate Python's dynamic nature. Interestingly, these two approaches employ fuzzing for distinct objectives: our tool enriches concolic execution with fuzzing to guide the test toward unexplored branches after symbolic downgrading occurs. In contrast, CrossHair treats fuzzing as a lightweight alternative to full concolic execution, with an aim to balance test coverage and test efficiency.

3. Methodology

In this section, we briefly recall the main algorithm of PyCT and describe how to employ PyCT in automatic unit testing. For the latter, we introduce two approaches: the first, called

Invoked Function Testing (IFT), performs targeted testing by directly modifying specific function invocations to elevate downgraded symbolic variables and enforce concolic testing. The second, called Dynamic Function Tracking (DFT), uses a profiler to monitor all function calls dynamically, enabling real-time tracing without needing to predefine each function for PyCT testing. DFT can be construed as a dynamic version of IFT: while IFT examines only pre-specified functions, DFT captures real-time function calls across the entire execution flow, automatically expanding test coverage to include unexpected or deeply nested functions without prior specification.

3.1. Object-Oriented Concolic Testing

We first revisit the concolic testing algorithm that serves as the foundation of PyCT, which elegantly integrates symbolic execution with Python's dynamic and object-oriented model. It was initially proposed by Ball and Daniel [6], [19], and has been adapted and further refined by various Python testing tools [7], [8], [17], [18]. The core idea behind the algorithm is to substitute basic Python data types (such as integers and strings) with *concolic objects* to maintain both the concrete and symbolic representations of a data variable. Each concolic object includes two components: a concrete value and a symbolic expression representing this value in terms of input variables. For example, an integer variable a might be represented as a concolic object $c_a = (2, x + 1)$, where 2 is the concrete value of a and $x+1$ is an expression depicting the relationship between a and an input variable x, namely, $a = x + 1$.

During testing, the algorithm uses concolic objects to track and update constraints during program execution (Figure 1). For each path executed, the algorithm maintains a path condition tree T and a queue Q of unexplored branches. The tree T records all explored paths by maintaining nodes with constraints encountered during execution. Simultaneously, Q stores formulas representing unexplored branches, which are derived by negating conditions in the current path. The algorithm uses these stored formulas to generate new inputs for subsequent runs, ensuring that unexplored paths are systematically covered. This iterative process of extending T and processing Q enables comprehensive exploration of the tested code: even in complex programs with nested conditionals, this process ensures that every condition is tested with different combinations of inputs.

To derive path constraints from concolic objects, the member functions of Python's data types must be overridden to support both symbolic and concrete computation. Consider our previous example: when an overridden member function $c_a.f$ operates on a concolic integer variable a, it performs two key updates. First, it updates c_a's concrete value 2 exactly as the original function $a.f$ would. Second, it transforms the symbolic expression $x + 1$ into a new expression that precisely captures what happens when $a.f$ is called with $a = x + 1$. This dual-update design enables the concolic tester to handle Python's diverse operations while preserving symbolic representations wherever feasible.

Unfortunately, not every function can be overridden to support symbolic expressions (e.g. when they involve operations or data structures that do not admit effective encoding in SMT theories [24]). Whenever the concolic engine encounters such unsupported functions or constructs, it automatically downgrades the concolic object to a concrete value. This fallback mechanism allows executions to continue without runtime exceptions even when full symbolic support is not available. Despite its flexibility, concolic object downgrading can lead to reduced symbolic coverage, as noted in the PyCT and CrossHair implementations [7], [8]. This limitation motivated the development of our techniques in this paper, which aim to minimize coverage loss through two key strategies: dynamically upgrading input values and independently testing unsupported operations.

3.2. Invoked Function Testing

As mentioned in the previous section, concolic testing may face challenges in handling unsupported operations, resulting in the downgrading of symbolic variables to concrete values. To overcome this, we can execute PyCT individually for all invoked functions. This adaptation restores symbolic variables in scenarios where they have been downgraded but subsequent code still requires testing.

More specifically, the Invoked Function Testing (IFT) method involves modifying the loader in Python's unit tests to inspect all functions within the tested module. This method utilizes the inspect package's getMembers function to check if the objects called within the tested module have function or class types. If the object at hand is of function type and any of its parameters has types int, float, str, or boolean (i.e. the types currently supported by PyCT), we create corresponding symbolic variables for the concrete variables and replace the function with a wrapped function that automatically uses concolic objects for PyCT execution when called. For objects of class types, we apply the same checking method as for functions and replace all functions within the class accordingly.

As a result, each invocation of a wrapped function triggers automatic PyCT testing, which ensures that the execution paths inside functions are also explored symbolically. This mechanism helps identify errors or vulnerabilities that may be missed if the functions are not tested in isolation, thereby achieving a higher level of code coverage than the ordinary unit-test method.

3.3. Dynamic Function Tracking

While IFT requires manual function wrapping, Dynamic Function Tracking (DFT) leverages Python's built-in profiler system for automatic, real-time monitoring of function calls and execution paths. This profiling capability provides

```
def isPalindrome(int x):
    y = 0
    z = x
    while z > 0:
        y = lshift(y)
            + z % 10
        z = rshift(z)
    if y == x:
        return True
    else:
        return False
def lshift(x):
    return x * 10
def rshift(x):
    return x // 10
```

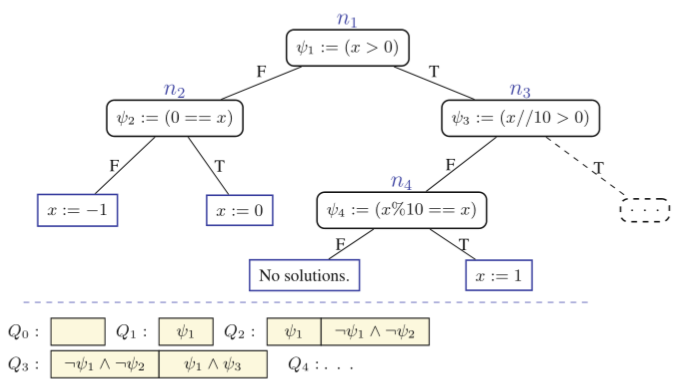

Figure 1. A concolic testing example from Chen et al. [7]. Left: a program for checking if x is a palindrome. Right: a snapshot of the tree T and queue Q in the concolic testing process on the program.

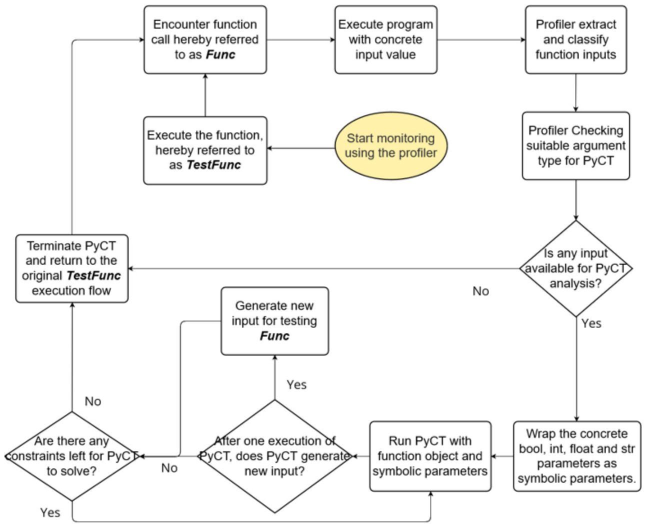
Encounter function call hereby referred to as *Func*
Execute program with concrete input value
Profiler extract and classify function inputs
Execute the function, hereby referred to as *TestFunc*
Start monitoring using the profiler
Profiler Checking suitable argument type for PyCT
Terminate PyCT and return to the original *TestFunc* execution flow
Is any input available for PyCT analysis?
No
Yes
Generate new input for testing *Func*
Yes
No
Are there any constraints left for PyCT to solve?
No
After one execution of PyCT, does PyCT generate new input?
Run PyCT with function object and symbolic parameters
Wrap the concrete bool, int, float and str parameters as symbolic parameters.
Yes

Figure 2. Flowchart of Dynamic Function Tracking

insights into the program's behavior at run time without requiring explicit code instrumentation.

We implemented a custom profiler embedded within program execution flow based on Python's profiling mechanism (i.e. using sys.setprofile), which enables us to capture detailed information about each function whenever the main program calls any functions (including function calls within these functions), such as the function name, function object, and the current input values used. Our profiler is designed to assess whether the function can be invoked independently, whether it requires input parameters, and whether any of these parameters are appropriate for PyCT testing. To illustrate, consider the situation when the profiler records a call to the updatecache function in linecache.py:

```
def updatecache(filename, module_globals=None)
```

Our tool detects two parameters for this function: a string variable filename and a module variable module_globals. Since string type is supported, this function is analyzable by PyCT. If this function has not been previously analyzed in the current test, our tool will mark filename as a concolic object and keep the concrete value of module_globals. The tool then passes the path of the function and the function object itself to PyCT for concolic testing. We outline the testing process using DFT in Figure 2.

During this process, the stack and frame mechanisms in Python play a crucial role: each time a function is called, a new frame is created and pushed onto the top of the call stack. This frame contains all the necessary information for the function's execution, including local variables, global variables, code objects, and instruction pointers. When the updatecache function executes, its frame is maintained in the stack, enabling the profiler to monitor and record the function calls and their execution paths in real-time. Once the function execution is complete, the frame is popped from the stack, and the control returns to the previous frame that called the function. This frame and stack mechanism allows the system to dynamically capture and analyze function calls, ensuring that PyCT can perform concolic testing on eligible functions.

Compared to the function invocation method described in Section 3.2, DFT has several advantages. First, DFT eliminates the need to wrap and wait for the functions and objects within the tested module to be called, since it dynamically examines the invoked functions. Moreover, DFT avoids the need to manually increase the levels of inspection (i.e. extending to record all functions within the objects called by the tested module, and subsequently the functions called within those objects, and so on). Indeed, DFT allows for the straightforward acquisition and examination of the currently executing function, regardless of its depth in the call stack.

Overall, integrating a profiler with PyCT systematically extends test coverage beyond the initial test cases: the profiler captures relevant functions, and PyCT produces new inputs based on runtime information to challenge their robustness

Algorithm 1 Dynamic Function Tracking with Fuzzing

```
Require:
    func: function object of the target function $func$
    path: full path of $func$
    args: list of argument values for $func$
    visited: set of full paths of the analyzed functions
Ensure:
    path is contained in visited
    The target function $func$ is analyzed by PyCT
 1: if path is in visited then
 2:     return
 3: end if
 4: Add path to visited
 5: for each arg in args do
 6:     if arg is of primitive type then
 7:         Upgrade arg to a concolic object
 8:     end if
 9: end for
10: Invoke PyCT with func, path, args
11: if args contains concrete values then
12:     repeat num_fuzz times
13:         Update concrete values in args using fuzzing
14:         Invoke PyCT with func, path, args
15:     end
16: end if
```

and security. Executing these functions with the new inputs uncovers hidden defects and enhances code path exploration, achieving higher line coverage and revealing vulnerabilities that more straightforward testing methods might miss.

3.4. Dynamic Function Tracking with Fuzzing

Fuzzing and concolic testing offer distinct approaches to uncovering software defects: while concolic testing systematically explores execution paths by combining symbolic and concrete inputs, fuzzing generates a wide range of random and mutated inputs to reveal edge cases through unpredictability. Our tool enables the integration of DFT with fuzzing, thereby enhancing DFT's real-time monitoring capabilities by injecting random inputs into each function call captured. This combination leverages DFT's detailed tracking of function calls with fuzzing's ability to uncover defects that may remain hidden from the vanilla DFT.

More specifically, the fuzzing-integrated DFT follows the same profiling and function-tracking setup as the standard DFT, with one key addition: when the profiler detects a function call, the fuzzer generates random inputs for non-primitive data types that concolic testing cannot directly cover. For example, in list, array, and tuple variables, the fuzzer may insert random values of supported types, such as str, float, int, and bool. These new inputs are then offered to PyCT for further concolic testing of the target function. This procedure is outlined in Algorithm 1, where a user-provided parameter num_fuzz specifies the maximal number of times the fuzzer is invoked to explore the function. To avoid unnecessary testing, our procedure records all functions that have already been analyzed by storing their paths in visited. In our experiment, we further reduce the number of fuzzy tests using a heuristic: we invoke PyCT after fuzzing only when the new input yields a return value different from the previous return values of the tested function. The flowchart of this procedure is given in Figure 3.

Conceptually, integrating fuzzing into DFT extends DFT's capabilities by exploring potential vulnerabilities that concolic testing alone might miss due to downgrading. While fuzzing is inherently random and may not systematically cover all paths, the randomness introduced by fuzzing can be harnessed within concolic execution to enhance systematic coverage analysis and broaden path exploration. Fuzzing's simplicity, flexibility, and ability to generate test inputs for unsupported types make it a promising addition to DFT. With effective guiding strategies to maximize input diversity and relevance, fuzzing can potentially strengthen vulnerability detection and provide a more comprehensive testing framework.

4. Empirical Evaluation

We implemented our dynamic testing approach as an extension of PyCT[1] and empirically evaluated its effectiveness from several perspectives:

- **Number of called packages and modules.** These two metrics represent how many packages and modules are used during the testing of the package. A higher count indicates a greater dependency on external libraries. If these libraries have any issues (e.g. version incompatibility, deprecation, or vulnerabilities), the program might fail to run properly. Additionally, using more imported packages can affect the portability of the program, as different systems and environments might support different libraries. If the required packages are unavailable in a particular environment, the program will not run. DFT can identify such errors by not only testing the package itself but also checking the security of the underlying code it relies on.
- **Overall lines executed.** This metric represents the code coverage for all Python scripts called during the testing process. It provides a holistic view of the total lines of code executed across the entire Python environment.
- **In-library lines executed.** This metric measures the code coverage specific to the library under test. It focuses on the effectiveness of the test in covering the target library's codebase.
- **Input number.** This denotes the number of inputs used during the execution of a unit test. For the baseline, it corresponds to the number of unit-test

1. The source code and scripts for this evaluation are available on https://github.com/incite-admin/pyct-profiler.

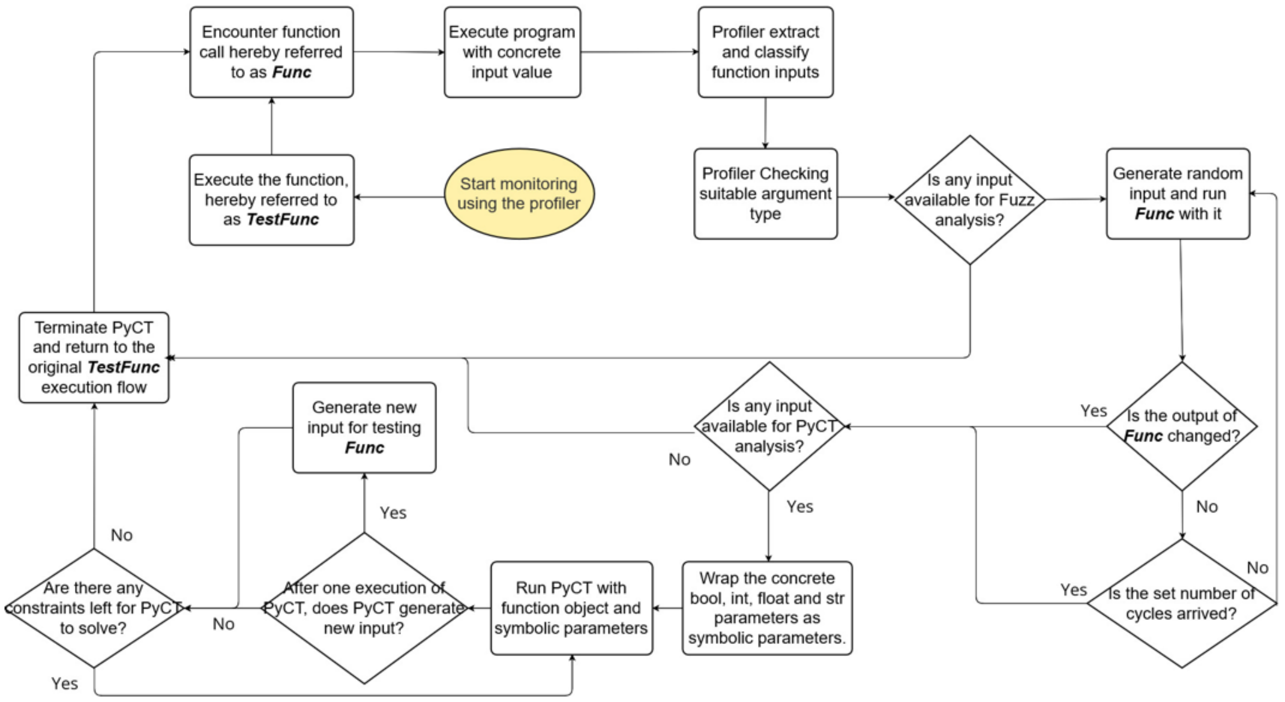
Encounter function call hereby referred to as ***Func***
Execute program with concrete input value
Profiler extract and classify function inputs
Execute the function, hereby referred to as ***TestFunc***
Start monitoring using the profiler
Profiler Checking suitable argument type
Is any input available for Fuzz analysis?
Generate random input and run ***Func*** with it
Terminate PyCT and return to the original ***TestFunc*** execution flow
Generate new input for testing ***Func***
Is any input available for PyCT analysis?
Yes
Is the output of ***Func*** changed?
No
Yes
Yes
No
No
Are there any constraints left for PyCT to solve?
No
After one execution of PyCT, does PyCT generate new input?
Run PyCT with function object and symbolic parameters
Wrap the concrete bool, int, float and str parameters as symbolic parameters.
Yes
Is the set number of cycles arrived?
No
Yes

Figure 3. Dynamic Function Tracking with Fuzzing

cases available in the library. This metric helps understand the extent of test-case utilization and input variety during testing. For DFT, it indicates how many additional inputs these methods can generate for testing.

- **PyCT invocations.** This metric counts the number of times PyCT was invoked during the testing process. It provides insight into the behavior and efficiency of PyCT in handling the test execution. Additionally, during the analysis of functions and the attempt to generate new inputs, PyCT relies on a solver to resolve constraints, which can result in either SAT (satisfiable) or UNSAT (unsatisfiable) or timeout (the solver fails to find a solution within the allotted time.). If the result is SAT, a new input is successfully generated and re-injected into the original function for execution. The outcome leads to either an execution success or a triggered error.
- **Execution success.** This refers to the number of successful executions when the generated test inputs are re-injected into the original function for testing.
- **Triggered error.** This indicates the number of errors triggered during the execution of the unit test. This metric is crucial for evaluating the robustness and reliability of the library when subjected to testing.
- **Timeout.** This indicates the frequencies of PyCT exceeding 600 seconds during the testing process. A higher value suggests that the constraints generated based on the branches within the function are more complex.

Our experiment compares these performance metrics of Python's `unittest` library and DFT,[2] with and without fuzzing, using various test suites. We address the following research questions in our evaluation:

- **RQ1: Exploration.** How does DFT perform compared to IFT and `unittest` regarding code coverage and checking dependency on external libraries?
- **RQ2: Error detection.** How does DFT perform compared to `unittest` regarding identifying errors related to external library dependencies and overall code robustness?
- **RQ3: Performance.** How efficient is DFT in generating additional inputs and handling complex constraints during testing?
- **RQ4: Fuzzing efficacy.** How does fuzzing benefit DFT in generating additional inputs and handling complex constraints during testing?

2. We did not conduct separate experiments for IFT, since the capability of IFT is strictly subsumed by DFT.

4.1. Benchmark

Our test suites include popular Python libraries and a neural network inference algorithm.

- **Jsonschema**: a package that describes the structure of JSON data, enabling developers to validate JSON objects against predefined schemas.
- **OpenAI**: this package serves as a standard for experimental purposes, primarily used for testing APIs and handling input prompt content in local applications.
- **Transformers**: a versatile library for natural language processing tasks, offering pre-trained models and tools for text generation, translation, and summarization. The baseline testing uses its built-in unit test cases.
- **Numpydoc**: a documentation generation tool for NumPy-style doc-strings, crucial for maintaining comprehensive and standardized documentation in scientific and analytical Python libraries. Its baseline testing also relies on built-in unit test test cases.
- **CNN Inference**: we tested a single invocation of a CNN model's predict function using pre-established data and model. This experiment evaluates DFT's capability to test an entire package from a single critical function call. The CNN model is implemented with Keras, and all numerical results are based on Keras as the target package for experimentation.

For each test suite, the experimental results are divided into three categories: Python's unittest library (as the baseline), DFT, and DFT+Fuzz. The baseline coverage is determined by executing the test cases included in the suite. The DFT coverage is assessed by running PyCT with the DFT enhancement. The DFT+Fuzz coverage is measured by running the DFT coverage combined with the fuzzing method. The results are outlined in Table 1.

4.2. Experimental Results

4.2.1. RQ1: Exploration. DFT demonstrated significant improvements in code coverage and dependency checking compared to the baseline unit tests across all benchmarks. In the Jsonschema and OpenAI benchmark, the numbers of packages and modules called and in-library lines covered all increased drastically with the application of DFT, indicating enhanced code coverage and more thorough checking of external library dependencies. The Transformers and Numpydoc packages also exhibited significant improvements in code coverage when tested with DFT. The CNN inference experiment, which had a single invocation of the predict function, also showcased DFT's effectiveness in maintaining testing standards even with minimal code and unit test cases.

Across all benchmarks, DFT consistently outperformed the baseline unit tests in terms of code coverage and dependency checking. The increased number of called packages and modules, along with the higher percentage of in-library lines covered, highlights DFT's effectiveness in enhancing test comprehensiveness and identifying potential issues arising from interactions between the tested package and its dependencies. These findings suggest that DFT is powerful for improving code coverage and dependency checking, regardless of the type of package being tested or the availability of comprehensive unit test cases. The ability to maintain high testing standards even with minimal initial inputs makes DFT particularly valuable in scenarios where traditional unit testing may be limited.

4.2.2. RQ2: Error detection. DFT significantly enhanced the identification of errors related to external library dependencies and improved overall code robustness compared to the baseline unit tests across all benchmarks. In the Jsonschema and OpenAI packages, DFT increased the number of triggered errors by orders of magnitude compared to the baseline unit tests. Similarly, the Transformers and Numpydoc packages saw substantial increases in triggered errors when tested with DFT.

The CNN inference experiment, which had an extremely limited baseline of a single predict function invocation, showcased DFT's ability to generate a large number of error-triggering inputs despite the scarcity of unit test cases. Interestingly, the number of triggered errors significantly exceeded the number of successful executions in this scenario.

Across all benchmarks, DFT consistently and often dramatically increased the number of triggered errors compared to the baseline unit tests. These results highlight DFT's effectiveness in uncovering errors related to external library dependencies and robustness issues that the baseline tests failed to identify. The CNN inference case further emphasizes DFT's capability to generate numerous error-triggering inputs even in scenarios where unit tests are extremely limited.

4.2.3. RQ3: Performance. DFT demonstrated exceptional capability in generating additional test inputs across all benchmarks, albeit at the cost of increased execution time due to the complexity of constraint handling. For example, in the tests of Jsonschema and OpenAI, DFT generated 1877 and 3067 additional inputs, respectively. However, Jsonschema incurred 3306 timeouts of 600 seconds each, and OpenAI 1410 timeouts caused by constraint solving. Similarly, while DFT generated 2416 and 5539 additional inputs for the Transformers and Numpydoc packages, respectively, the time cost associated with these additional inputs was also substantial. The CNN inference experiment stood out as an exception, where DFT efficiently generated additional inputs without encountering any timeouts, possibly due to the characteristics of the tested package (Keras).

In summary, while DFT excels in generating additional test inputs, the efficiency of this process is highly dependent on the complexity of the constraints encountered during testing. The trade-off between the benefits of increased input

TABLE 1. EXPERIMENTAL RESULTS ON TESTING **JSONSCHEMA**, **OPENAI**, **TRANSFORMERS**, **NUMPYDOC**, AND **CNN INFERENCE**

	Jsonschema			**OpenAI**			**Transformers**		
Metric	unittest	DFT	DFT+Fuzz	unittest	DFT	DFT+Fuzz	unittest	DFT	DFT+Fuzz
package numbers	8	37	37	20	67	68	47	62	64
module numbers	50	123	125	144	399	405	382	550	554
overall lines executed	4090	45768	46440	31049	97973	98003	120,710	280,218	280,302
in-library lines executed	3125	5684	5803	666	2224	2243	31,194	45,906	45,920
PyCT invocations	N/A	9213	9130	N/A	13754	13818	N/A	10,169	10,212
input numbers	473	2350	2346	53	3120	3144	2,501	4,439	4,461
execution successes	471	1069	1066	44	2700	2711	2,023	3,600	3,618
triggered errors	2	1281	1280	9	420	433	478	839	843
timeouts	N/A	3306	3304	N/A	1410	1415	N/A	5,730	5,751

	Numpydoc			**CNN Inference**		
Metric	unittest	DFT	DFT+Fuzz	unittest	DFT	DFT+Fuzz
package numbers	29	48	48	28	52	54
module numbers	244	295	297	180	311	315
overall lines executed	44,203	96,180	96,209	122,016	202,204	202,821
in-library lines executed	2,194	2,593	2,597	14,921	17,012	17,354
PyCT invocations	N/A	15,199	15,226	N/A	3,435	3,468
input numbers	253	5,792	5,784	1	2,434	2,448
execution successes	250	3,560	3,556	1	676	678
triggered errors	3	2,232	2,228	0	1,758	1,770
timeouts	N/A	237	241	N/A	0	0

generation and the associated time costs should be carefully considered when applying DFT in practice.

4.2.4. RQ4: Fuzzing efficacy. We assessed the integration of fuzzing into DFT to evaluate its effect on unit tests. In our experiment, we simply generated legitimate random values of supported types, namely int, float, str, and boolean. The results across different benchmarks (listed in the DFT+Fuzz columns) reveal that these random inputs generally increased the number of inputs, such as in the OpenAI and Numpydoc tests. However, in the Jsonschema test, the input count actually decreased slightly when fuzzing was enabled. This suggests that the random guessing in our fuzzing approach is not consistently effective. In addition, while fuzzing occasionally contributed to exploring new paths, it also led to a slight increase in timeouts, as seen consistently across benchmarks like Transformers and Numpydoc. This increase in timeouts indicates that the random inputs generated by the basic fuzzing technique might not always lead to meaningful exploration and could, in some cases, prolong execution without yielding new paths or constraints.

While the improvements in coverage from fuzzing were modest in our experiment, this is likely due to the limitations of the random guessing method employed. Literature suggests that more advanced techniques, such as mutation-based or coverage-guided fuzzing (cf. [20], [21]), could potentially enhance the effectiveness of DFT. Future work will focus on integrating these more sophisticated fuzzing methods to achieve a balance between input diversity and execution efficiency.

4.2.5. Summary. Overall, DFT significantly increases code line coverage with minimal unit test inputs, effectively tests dependencies on external libraries, and generates inputs that trigger numerous errors and assertions in both the tested code and its dependent libraries. The integration of fuzzing also broadens the testing scope. The improvements were particularly noticeable in packages with complex dependencies, such as Transformers and CNN inference models, where DFT significantly increased the number of triggered errors and exposed hidden vulnerabilities. However, the benefits of applying DFT can come at the cost of longer execution times and more timeouts. These findings suggest that while DFT offers capabilities in improving code coverage and error detection, careful consideration is crucial for practical implementation in software development workflows.

5. Threats to Validity

Our dynamic testing approach aims to mitigate the limitations of traditional concolic testing. However, certain threats to its validity arise from the operational assumptions underlying DFT, which may impact its effectiveness in practice.

A primary threat to the validity of DFT lies in its reliance on dynamic profiling to extract constraints from external functions and unsupported operations. The accuracy of the symbolic constraints derived during this phase is contingent on the comprehensiveness of the profiling data. If the profiling fails to capture the full range of possible behaviors due to insufficient execution traces or limited runtime data, the generated constraints may misrepresent the true behavior of the functions. Additionally, certain external functions may exhibit dynamic behavior influenced by external states, such as network conditions or system time, which makes reliable profiling particularly challenging.

Another concern is the potential for performance overhead introduced by the dynamic profiling process. While DFT attempts to enhance path coverage by addressing symbolic downgrading, the added computational cost of profiling and processing external calls may limit its scalability, particularly in systems with extensive use of external libraries. Finally, Python's dynamic and flexible features, such as metaprogramming and runtime modifications, may lead to scenarios where DFT struggles to maintain symbolic representations effectively. The success of DFT in resolving unsupported operations depends heavily on its ability to handle the diverse and evolving nature of Python's libraries and external APIs. Incomplete or outdated handling of such constructs could reduce the applicability and robustness of the technique.

6. Conclusion

In this study, we have explored the efficacy of integrating concolic testing with runtime profiling in software testing. By dynamically monitoring function calls and wrapping the called functions, we extended the capabilities of the PyCT framework in scenarios involving external dependencies on Python libraries. The experimental results validate our approach's ability to enhance code coverage and error detection, surpassing traditional unit testing methods. However, the trade-off between increased coverage and longer execution times remains challenging.

Our future research will pursue three key directions. First, we plan to incorporate more advanced fuzzing techniques such as mutation-based and coverage-guided fuzzing to improve input generation and path exploration. Second, we will develop a graphical interface that visualizes the testing process in real-time, enabling developers to better understand test coverage and track error detection. Finally, we will extend our concolic engine to handle a broader range of data structures and operations.

References

[1] Hong Zhu, Patrick AV Hall, and John HR May. Software unit test coverage and adequacy. *Acm computing surveys*, 29(4):366–427, 1997.

[2] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing, 2018.

[3] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, feb 2013.

[4] Lucilia Y. Araki. and Leticia M. Peres. A systematic review of concolic testing with aplication of test criteria. In *Proceedings of the 20th International Conference on Enterprise Information Systems - Volume 2: ICEIS*, pages 121–132. INSTICC, SciTePress, 2018.

[5] Koushik Sen. Concolic testing. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, pages 571–572, 2007.

[6] Thomas Ball and Jakub Daniel. Deconstructing dynamic symbolic execution. In *Dependable Software Systems Engineering*, pages 26–41. IOS Press, 2015.

[7] Yu-Fang Chen, Wei-Lun Tsai, Wei-Cheng Wu, Di-De Yen, and Fang Yu. Pyct: A python concolic tester. In Hakjoo Oh, editor, *Programming Languages and Systems*, pages 38–46, Cham, 2021. Springer International Publishing.

[8] Phillip Schanely. Crosshair - an analysis tool for python that blurs the line between testing and type systems. https://github.com/pschanely/CrossHair, 2017. [Online; accessed 20-Oct-2024].

[9] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[10] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[11] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[12] Chang Liu, Xiwei Wu, Yuan Feng, Qinxiang Cao, and Junchi Yan. Towards general loop invariant generation via coordinating symbolic execution and large language models. *arXiv preprint arXiv:2311.10483*, 2023.

[13] Zexu Wang, Jiachi Chen, Yanlin Wang, Yu Zhang, Weizhe Zhang, and Zibin Zheng. Efficiently detecting reentrancy vulnerabilities in complex smart contracts. *Proceedings of the ACM on Software Engineering*, 1(FSE):161–181, 2024.

[14] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation*, volume 8, pages 209–224, 2008.

[15] Corina S Păsăreanu and Neha Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, pages 179–180, 2010.

[16] Florent Saudel and Jonathan Salwan. Triton: A dynamic symbolic execution framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes*, pages 31–54, 2015.

[17] Björn I. Dahlgren. Pysym documentation. https://pythonhosted.org/pysym/, 2016. (2016).

[18] D. Barsotti, A.M. Bordese, and T. Hayes. Pef: Python error finder. In *Selected Papers of the XLIII Latin American Computer Conference (CLEI)*, volume 339 of *Electronic Notes in Theoretical Computer Science*, pages 21–41. Elsevier, 2017.

[19] Thomas Ball and Jakub Daniel. Deconstructing dynamic symbolic execution. In Maximilian Irlbeck, Doron A. Peled, and Alexander Pretschner, editors, *Dependable Software Systems Engineering*, volume 40 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 26–41. IOS Press, 2015.

[20] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2021.

[21] Xiaoqi Zhao, Haipeng Qu, Jianliang Xu, Xiaohui Li, Wenjie Lv, and Gai-Ge Wang. A systematic review of fuzzing. *Soft Comput.*, 28(6):5493–5522, oct 2023.

[22] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Network and Distributed System Security Symposium*, volume 16, pages 1–16, 2016.

[23] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, 2018.

[24] Aaron R Bradley and Zohar Manna. *The calculus of computation: decision procedures with applications to verification*. Springer Science & Business Media, 2007.