Real-World Fault Detection for C-Extended Python Projects with Automated Unit Test Generation

Lucas Berg*⊥, Lukas Krodinger†⊥, Stephan Lukasczyk‡, Annibale Panichella§,
Gordon Fraser†, Wim Vanhoof* and Xavier Devroey*
*NADI, University of Namur, Namur, Belgium
†University of Passau, Passau, Germany
‡JetBrains Research, Munich, Germany
§Delft University of Technology, Delft, Netherlands

***Abstract*—Many popular Python libraries use C-extensions for performance-critical operations allowing users to combine the best of the two worlds: The simplicity and versatility of Python and the performance of C. A drawback of this approach is that exceptions raised in C can bypass Python's exception handling and cause the entire interpreter to crash. These crashes are real faults if they occur when calling a public API. While automated test generation should, in principle, detect such faults, crashes in native code can halt the test process entirely, preventing detection or reproduction of the underlying errors and inhibiting coverage of non-crashing parts of the code. To overcome this problem, we propose separating the generation and execution stages of the test-generation process. We therefore adapt PYNGUIN, an automated test case generation tool for Python, to use subprocess-execution. Executing each generated test in an isolated subprocess prevents a crash from halting the test generation process itself. This allows us to (1) detect such faults, (2) generate reproducible crash-revealing test cases for them, (3) allow studying the underlying faults, and (4) enable test generation for non-crashing parts of the code. To evaluate our approach, we created a dataset consisting of 1 648 modules from 21 popular Python libraries with C-extensions. Subprocess-execution allowed automated testing of up to 56.5 % more modules and discovered 213 unique crash causes, revealing 32 previously unknown faults.**

***Index Terms*—Test Generation, Python, Fault Detection, Foreign Function Interface, C-Extension, Subprocess**

I. INTRODUCTION

Python has emerged as one of the most widely used programming languages in scientific computing [1], data analysis, and artificial intelligence, primarily due to its simplicity and extensive ecosystem of libraries. However, as an interpreted language, Python may not always provide the performance necessary for computationally intensive tasks. To mitigate these performance bottlenecks, Python's *foreign function interface* (FFI) allows it to interface with compiled languages, particularly C/C++. Many Python libraries, especially in the scientific computing and machine-learning domains, like NUMPY [2], PANDAS [3], and TENSORFLOW [4] use FFIs.

However, the FFI can lead to a crash of the Python interpreter if misused. For example, Figure 1 shows the function `idd_reconid` from the scientific computing library SCIPY [5]. It is written in `Cython` [6], a superset of Python that compiles to C and allows developers to write C-extension modules using a Python-like syntax. After compilation, developers can import `idd_reconid` from its C-extension module and call it from Python code as if it were a regular Python function. This ability of the Python interpreter to import and invoke functions written in other languages is made possible by Python's FFI. However, the function `idd_reconid` does not validate its first parameter: passing a one-dimensional array instead of a NUMPY matrix can result in a *segmentation fault* at line 4. The crash occurs because the Python interpreter executes native code, thereby bypassing Python's built-in memory-safety mechanisms. Unlike Python, which provides automatic memory management and bounds checking, C requires manual memory allocation and permits direct memory access. In this example, SCIPY enables a Cython optimisation that disables bounds checking to improve performance. As a result, indexing operations follow C semantics, causing *segmentation faults* instead of raising `IndexErrors` [7]. This example illustrates only one type of fault that can arise from misusing Python's FFI; in addition to *segmentation faults*, other low-level errors such as *bus errors*, *memory leaks*, and *floating-point exceptions* can also occur. All these faults are particularly severe because they can cause the Python interpreter to crash, which poses a threat to software reliability.

To mitigate this risk, diagnosis tools were developed, especially to detect *memory leaks* [8], using binary instrumentation [9], static analysis [10], and dynamic analysis for Java [11]

```
1   # cython: boundscheck=False
2
3   def idd_reconid(B, idx, proj):
4       cdef int m = B.shape[0], krank = B.shape[1]
5       cdef int n = len(idx)
6       approx = np.zeros([m, n], dtype=np.float64)
7
8       approx[:, idx[:krank]] = B
9       approx[:, idx[krank:]] = B @ proj
10
11      return approx
```

Figure 1: The `idd_reconid` function from SCIPY, which is included in a C-extension module, can cause a *segmentation fault* at line 4 if the argument `B` is not a valid NUMPY matrix.

978-1-5386-5541-2/18/$31.00 ©2026 IEEE. This is the authors' version. The final version is published in *Proceedings of the 19th IEEE International Conference on Software Testing, Verification and Validation (ICST)*.

⊥The first two authors contributed equally to this work.

and Python [12]. However, these tools do not automatically find issues and require manual intervention: developers must already have a failing test or know how to reproduce an issue to apply a memory-leak diagnosis tool. This leaves developers to manually locate faults and reverse-engineer the conditions that trigger them, which is time-consuming and error-prone.

To alleviate developers' workload and manage effort and costs, automated unit test generation tools have been introduced. Notable examples of existing tools include RANDOOP [13] and EVOSUITE [14] for Java or PYNGUIN [15] for Python. These tools typically examine a *subject under test* (SUT), such as a Java class or Python module, and use meta-heuristic search algorithms to generate test cases. The objective is to create inputs that trigger the SUT's routines (i.e., functions, methods, and constructors) in a way that optimises a specific fitness metric, most commonly branch coverage. During coverage optimisation, the tools iteratively generate, modify, and execute tests while measuring their coverage. Consequently, gradually more parts of the SUT are tested. Exceptions triggered during execution may indicate faults in the SUT, which holds for search- and LLM-based test generation approaches and was previously explored for Java [16], [17], but not yet for Python.

PYNGUIN [15] generates tests for Python and, like any other Python-based tool, requires an interpreter to run and execute generated tests. However, a *segmentation fault* or a similar C error uncaught by Python's exception handling can crash the Python interpreter, which in turn terminates the test-generation process and thus prevents the detection of the underlying fault causing the crash. In Java, where FFIs are less common, tools like EVOSUITE did not require process isolation in past studies. In contrast, Python's extensive use of FFIs means that interpreter crashes due to C errors pose a significant challenge for automated test generation tools like PYNGUIN, hindering developers to detect, quantify and investigate bugs in C-extended projects.

By executing the SUT in a separate subprocess rather than only in a separate thread, we isolate the test generation process from the SUT's execution and thereby overcome this limitation of PYNGUIN. The architectural separation acts as a sandbox, protecting the test generator itself. If a test case triggers a fatal error that cannot be caught from its Python invocation point, like a *segmentation fault* in the SUT, only the subprocess terminates, leaving the main search process unharmed. Our approach monitors these subprocesses and detects crashes. Upon detection, the corresponding test case that triggered the fault is exported as a PYTEST [18] test case. Re-execution ensures that the test case is reproducible, so developers can use it to debug and fix underlying issues. Overall, our approach has four advantages compared to running PYNGUIN without subprocess-execution: (1) it allows PYNGUIN to detect crash-revealing faults in the SUT, (2) it enables the generation of crash-revealing test cases, (3) it enables the study of the underlying faults, and (4) it makes it possible to continue the test-generation process even if the SUT crashes. These advantages come with a drawback: subprocess-execution requires more execution time than threaded-execution due to process communication overhead. We examine when subprocess execution is beneficial and present multiple strategies to automate the decision of which execution strategy to use. While spawning processes is not difficult, and done in fuzzing [19], it is challenging in the context of test generation because of the code instrumentation for fitness calculations. This, however, is crucial to allow PYNGUIN to explore the SUT without the need to manually create data providers, as in fuzzing. Previous studies on PYNGUIN [20], [15], [21], [22] did not focus on Python modules that use the Python FFI. We address this gap by explicitly targeting such modules. Although our work is grounded in the Python ecosystem, it tackles the broader challenge of robustly testing software that integrates native code via an FFI.

To evaluate our approach, we executed PYNGUIN with and without subprocess-execution on a large dataset of 1 648 Python modules from 21 popular libraries using C-extensions. We created this dataset as we are not aware of any existing datasets of popular real-world Python projects with C-extensions. Using subprocess-execution, we automatically generated 120 176 tests revealing 213 crash causes. We manually analysed these to identify 32 previously unknown faults. For example, crash-revealing tests raising *segmentation faults* allowed us to discover that the `idd_reconid` function of SCIPY, shown in Fig. 1, does not check its parameters properly. To validate our findings, we reported these bugs to the respective development teams.

In detail, the contributions of this paper are:

- We introduce a new subprocess-execution for PYNGUIN that allows generating test cases that can reveal faults.
- We create a new dataset of 1 648 Python modules from popular libraries that use C-extensions.
- We conduct a large-scale study of the faults detected by using PYNGUIN with subprocess-execution on the dataset.
- We identify 32 previously unknown faults in the libraries using the generated test cases.

Our evaluation demonstrates that the subprocess-execution of PYNGUIN is effective in automatically revealing faults in Python libraries that use C-extensions. We provide additional artefacts consisting of datasets, tool images, result data, and analysis scripts to Zenodo to allow for future use [23].

II. BACKGROUND

Our approach is based on unit test generation for Python programs and explicitly tailored to Python modules using the Python *foreign function interface* (FFI) to execute C/C++ code.

A. Foreign Function Interfaces

A *foreign function interface* (FFI) is a mechanism that allows code written in one programming language to call functions and access data structures written in another. The concept is not specific to Python [24], [25], [26], but it is particularly relevant to this language, as it allowed its ecosystem to expand into areas that usually require fast programming languages, such as artificial intelligence and data analysis. By interfacing with C/C++ code, many Python libraries offer a user-friendly and expressive interface while still delivering high performance.

This is mainly what makes Python the most widely used programming language in scientific computing [1].

There are several mechanisms for creating FFI bindings in Python, each with its trade-offs in terms of ease of use, performance, and portability [6], [27], [28], [29], [30]. However, their use introduces challenges as they rely on native code. For instance, while Python's exception system is designed to handle runtime errors via `try-except` blocks, it cannot intercept failures that originate in native code such as *segmentation faults*. While the Python `faulthandler` [31] module can dump the Python stack trace when such faults occur, it does not prevent the interpreter from crashing. Importantly, *segmentation faults* are merely one example from a broad spectrum of native faults. Other fatal issues include *memory leaks*, *illegal instructions*, *bus errors*, and *native assertion failures*. These faults are particularly critical because they undermine the reliability of any Python application and can lead to security vulnerabilities such as remote code execution. It is therefore essential to rigorously test code that uses the FFI bindings.

B. Unit Test Generation for Python

Creating strong test suites is essential for minimising the risk of software failures. However, the process of writing test cases manually can be labour-intensive. To address this, various automated approaches for generating unit tests have been developed [32]. Some methods rely on random generation, potentially augmented by feedback from previous executions to guide future test creation [33]. Other techniques employ *search-based software engineering* (SBSE) techniques, such as evolutionary algorithms, which seek to optimise test cases according to a defined fitness function [34], such as branch coverage. For an in-depth discussion of search-based test generation, we refer to the extensive literature on this topic [35], [36]. In recent years, research has explored the usage of generative models [37] and *large language models* (LLMs) for test generation [38], [39], [40], [22], [41], [42], [43], [44], [45], [46], [47], [48], [49], [50]. However, regardless of the used approach, the generated tests must be run against the SUT to measure their effectiveness, and therefore all approaches can be affected by potential crashes due to FFIs misuse. Previous studies [20], [15], [21], [22] did not consider code that uses FFI bindings.

PYNGUIN [15] is a hybrid state-of-the-art test-generation framework specifically designed for the Python programming language. It incorporates multiple SBSE test-generation strategies, including feedback-directed random test generation [33] and the DynaMOSA [51] evolutionary algorithm, as well as LLM-based and hybrid test-generation strategies such as CodaMosa [22]. A fundamental requirement for all strategies is the ability to execute generated tests against the SUT. During this execution, PYNGUIN measures the code coverage achieved by the generated test cases and uses the feedback on their effectiveness to further improve them, aiming to produce test cases with high coverage. However, PYNGUIN faces challenges when the execution of a SUT triggers a C error, such as a *segmentation fault*. Such errors cause the Python interpreter, which also runs PYNGUIN itself, to crash.

III. Crash-Revealing Test Generation

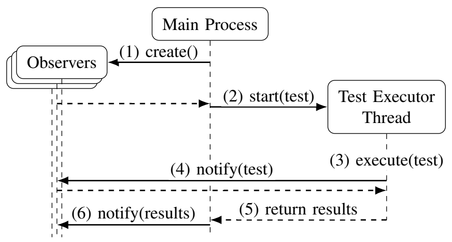
In situations where a SUT could cause the Python interpreter to crash, the effectiveness of PYNGUIN is limited. We developed subprocess-execution to overcome this. It improves PYNGUIN's robustness and allows it to create crash-revealing tests.

A. Test-Execution Model

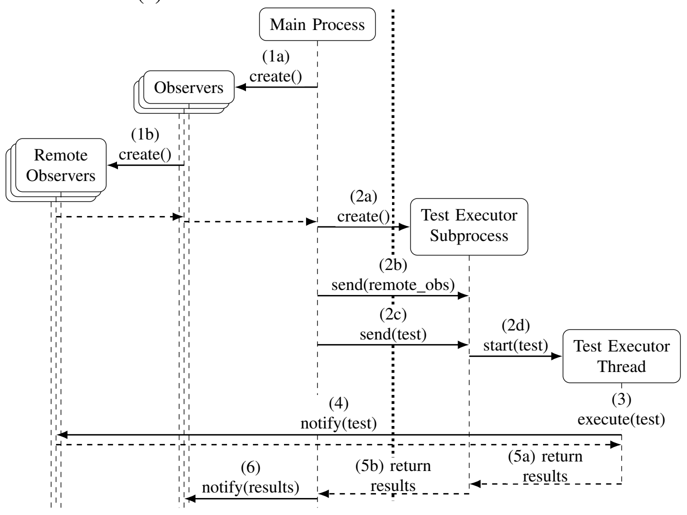
As the subprocess-execution is based on PYNGUIN's existing threaded-execution, we first describe this original execution model. The test generation is an iterative feedback loop, orchestrated by an observer system responsible for tracking and collecting diverse information, such as coverage, executed bytecode, or assertion results during test execution. Figure 2a shows the original execution model of PYNGUIN. At the start of the test generation, PYNGUIN initialises a set of observers (1) that monitor the test-generation process and collect relevant information. For each test case execution PYNGUIN *starts* a *Test Executor Thread* (2), which executes the test case (3) in a separate thread to enable control of the test-case execution time. Additionally, observers responsible for collecting test-execution results are handed over. During the execution of the test case, the SUT is monitored through on-the-fly bytecode instrumentation checking for achieved fitness. The instrumentation injects hooks into the subject's code (not shown in the figure), which allows observers to collect relevant execution data, including fitness values, executed bytecode, and assertion outcomes (4). These data are then combined into *results* and sent back to the main thread (5), where they are used to guide the test-generation process. Observers track metadata for the stopping conditions, such as the current iteration of the genetic algorithm, and some general data that must persist between iterations, such as inferred return types (6).

B. Subprocess-Execution

To address critical issues encountered during test-case generation, such as *segmentation faults*, we propose an architectural shift in PYNGUIN's test-execution model: the possibility to execute test cases in isolated subprocesses. This approach effectively mitigates FFI-related errors by leveraging operating system-level process management (instead of application-level thread management). However, this new architecture introduces some challenges and overheads. Firstly, this approach is slower, as spawning a subprocess is more costly than spawning a thread because it requires to start a new Python interpreter instead of allocating a new thread in the same process, which is a much lighter operation. This is aggravated by the impossibility to use shared memory between processes, requiring an additional serialisation step to send data between them. Secondly, PYNGUIN's architecture is heavily based on the observer pattern, which causes challenges because tests running in subprocesses need to be observed from the main process. We investigate the performance overhead in Section IV-E.



(a) Pynguin's threaded-execution model



(b) Pynguin's subprocess-execution model

Figure 2: The two execution models' execution sequences

1) Architecture Refactoring: To add subprocess-execution to Pynguin's execution model we decoupled the observers from the test-execution threads, as the original threaded-execution system had a single class that conflated responsibilities by containing methods that were executed in both the test-execution thread and the main thread. While this design was viable in a threaded environment due to shared memory, it relied on direct state access, which rendered observer instances non-serialisable. Serialisability, however, is a prerequisite for transferring objects between processes and is thus essential for subprocess-execution. Consequently, the original design was incompatible with process-based isolation, which requires explicit communication across separate memory spaces.

The new model divides observers into two categories. *Main Process Observers* run in the main process and monitor high-level information that is not directly related to test execution, such as the current iteration of the genetic algorithm or the best test cases found so far. *Remote Observers* must be sent to the test-executor subprocess and run there to collect test execution results, such as coverage information or assertion results. With subprocess-execution, as illustrated in Figure 2b, the *Main Process* creates all observers—both remote (1b) and main process observers (1a)—and creates a *Test Executor Subprocess* (2a). The *Main Process* sends the remote observers (2b) and the test case (2c) to be executed to the *Test Executor Subprocess* which then starts a *Test Executor Thread* (2d) to execute the test case (3) similar to threaded-execution. We still require the thread for fine-grained control over the test execution, e.g., to enforce timeouts. After creating the *results* using the remote observer data (4), they are sent back to the main process (5a, 5b) and used by the main process observers (6). Transferring remote observers and test cases between processes requires serialisation. For this purpose, we use the dill [52] and multiprocess modules from Pathos [53], [54], [55]. In short, we added the subprocess as an additional intermediate layer between the main process and the test-execution thread. In case of a critical C error only the respective subprocess might crash, leaving the main process, which runs Pynguin, intact.

2) Automatic Execution-Mode Selection: Using subprocess-execution is more robust when testing SUTs that rely on FFI code, but it also incurs a runtime overhead compared to threaded-execution. As Section IV shows, subprocess-execution is preferable when testing SUTs that use FFI code, while threaded-execution is faster for pure-Python SUTs. When running Pynguin, the user may not know whether a particular SUT module uses the FFI and thus might require subprocess-execution or not. We added three approaches to automatically decide which execution mode to use.

Heuristic. The heuristic execution mode automatically selects the optimal execution strategy by detecting FFI code during an initial analysis of the SUT. The detection mechanism first inspects the file extension of each imported module, identifying `.so` and `.pyd` files as C-extensions. For `.py` files, it checks for a pure Python implementation using `inspect.getsource` [56], which fails on non-pure-Python modules. If Pynguin detects C-extensions, it uses subprocess-execution; otherwise, it defaults to threaded-execution. A whitelist permits specific built-in modules, such as `sys` and `math`, to use threaded-execution despite their underlying C implementations.

Restart. The restart execution mode uses the faster threaded-execution and restarts the search with subprocess-execution when there is a crash. To achieve this, we added a master-worker architecture to Pynguin. When activated, Pynguin acts as a master process that in turn creates a worker in a subprocess to do the test generation. If the worker crashes in threaded-execution due to a critical C-error, the master adjusts the search time by subtracting used time, enables subprocess-execution and restarts the search with a new worker. While this introduces another subprocess layer, there are only two communication steps between the master and worker—one at the worker start and one when the worker finishes. Thus, this does not result in a notable runtime overhead.

Combined. When combining heuristic and restart, Pynguin does not use threaded-execution as initial execution mode but decides on it using the heuristic. Upon a crash, the search is restarted with subprocess-execution.

C. Revealing Faults

By preventing Pynguin from crashing, subprocess-execution enables the generation of test cases for modules that use C/C++ code. We extended Pynguin to export the test cases that cause a crash without subprocess-execution separately from the final test suite. Those crash-revealing test cases do not contribute to coverage, but help in reproducing and debugging the crash. We later execute each exported crash-revealing test case to record the error code which caused the crash.

Crash-Revealing tests may not be reproducible due to non-deterministic behaviour or internal state of the SUT. When reporting a crash, it is important to provide a reproducible example to help developers isolate and fix the root cause. To address this, we introduced a re-execution step into PYNGUIN. This step checks whether the generated crash-revealing test case can still trigger the crash.

IV. EVALUATION

We introduced subprocess-execution as described in the previous section. To evaluate its effectiveness, we ask:

RQ1: *How many* PYNGUIN *crashes can be avoided by using subprocess-execution?*

By preventing PYNGUIN from crashing, subprocess-execution allows generating crash-revealing tests which detect faults in the SUT. Since multiple crash-revealing tests can be generated for the same fault, we group the tests based on the last Python statement called before the crash occurred and the exit code of the crash-revealing test. We define the last statement as *crash cause*. To evaluate how well subprocess-execution performs in detecting real-world faults, we ask:

RQ2: *How many unique crash causes can be identified using the crash-revealing tests generated by subprocess-execution?*

While subprocess-execution allows for testing modules that would otherwise cause PYNGUIN to crash, resulting in 0 % coverage, it also introduces a performance overhead due to the subprocess creation and inter-process communication. To examine the overall impact of this on code coverage, we ask:

RQ3: *What is the impact of using subprocess-execution on achieved branch coverage?*

A. Evaluation Setup

We conducted an empirical evaluation as follows to answer our research questions.

1) Subjects: Datasets previously used to evaluate PYNGUIN did not focus on Python modules that use C-extensions [20], [15], [21], [22]. Thus, we created a new dataset, named DS-C.

DS-C: Table I shows an overview of DS-C, which contains 1 648 modules from 21 popular Python libraries that use C-extensions. We used CLOC [57] to measure the size of the used modules in *lines of code* (LoC) and the GITHUB API [58] to determine the size of the projects in megabytes for Python, C/C++, and the percentage of C/C++ code.

The dataset was created as follows: (i) we collected the 500 most popular Python libraries from the *Python Package Index* (PYPI) according to the TOP PYPI PACKAGES [59], [60]. (ii) For each project, we fetched metadata from PYPI and used the GITHUB API to identify its GitHub repository and the latest tags. Since Git tags often correspond to release versions on PYPI, we matched the version tag in the PYPI metadata with those from GitHub to identify the exact commit used for the release. (iii) As we require access to the source code to

Table I: Projects, modules per project (M.), version tag, SUT modules size (min, mean, max) in lines of code (LoC), and project size in megabytes (MB) for Python, C/C++ code and the percentage of C/C++ code in the project.

Project	M.	Tag	SUT Size (LoC)			Project Size (MB)		
			min	*mean*	*max*	*Python*	*C/C++*	*%*
cffi	13	1.17.1	65	468	1294	1.09	0.584	34.8
coverage	38	7.8.0	35	234	787	1.50	0.047 5	2.31
debugpy	18	1.8.14	40	211	782	6.23	0.101	1.50
distlib	11	0.3.9	107	555	1373	1.34	0.034 2	2.48
msgpack	1	1.1.0	79	376	673	0.081 8	0.050 8	30.6
mypy	4	1.15.0	20	304	803	5.74	0.282	4.68
numba	236	0.61.2	9	404	4791	10.2	0.989	8.81
numpy	83	2.2.5	7	249	2947	10.8	6.62	37.2
pandas	167	2.2.3	10	548	5199	20.0	0.338	1.53
pycparser	10	2.22	42	589	2023	0.510	0.073 0	12.5
pygments	115	2.19.1	19	370	3134	4.48	0.186	2.52
pymongo	50	4.12.1	14	352	1581	5.90	0.177	2.90
pytz	4	2025.2	86	129	208	0.104	0.257	35.2
rapidfuzz	6	3.13.0	66	146	374	0.429	0.400	38.4
scipy	8	1.15.2	14	359	1006	19.3	6.85	22.5
shapely	24	2.1.0	8	100	447	1.08	0.260	18.9
simplejson	3	3.20.1	31	217	511	0.167	0.104	38.4
tensorflow	745	2.19.0	6	232	2683	43.7	97.3	56.6
torch	92	2.7.0	8	227	2310	66.0	41.1	35.9
watchdog	15	6.0.0	20	139	586	0.337	0.031 4	8.50
wrapt	5	1.17.2	43	184	443	0.287	0.098 6	25.5

analyse the faults revealed by PYNGUIN, we filtered out projects that are not hosted on GitHub or where the tag could not be matched with a commit. (iv) In some cases, multiple PYPI packages reference the same GitHub repository (e.g., when a single project distributes multiple related components such as *tool-core* and *tool-extension*). We removed such duplicates based on identical GitHub URLs. (v) To focus on projects that use C-extensions, we queried the GITHUB API for each project's language composition and retained only those with at least 1 % of C or C++ code. (vi) We searched for all public (i.e., modules that do not start with an underscore) and non-test (i.e., modules that do not contain `test` in the name or package name) modules and added them to the dataset. (vii) We removed all modules that do not contain any code for PYNGUIN to test (i.e., modules that have no public classes or functions), modules that cannot be imported by PYNGUIN (e.g., due to missing dependencies or import errors) and all modules that require coroutines, which PYNGUIN does not support. In total, we removed 1 005 modules from the dataset due to these constraints. (viii) Some modules are trivial to test. For instance, if a module contains only a single function without branching, PYNGUIN often achieves 100 % branch coverage during the initial random generation phase, regardless of the execution mode (threaded-execution or subprocess-execution). Those trivial modules are neither interesting for measuring the performance of subprocess-execution nor for generating crash-revealing tests. Aligning with previous work [22], we removed 95 trivial modules where PYNGUIN achieved 100 % branch coverage within 1 min.

Our final dataset consists of 1 648 modules from 21 popular Python libraries that use C-extensions. Previous work on PYNGUIN used datasets with 104 to 486 modules [20], [15], [21], [22]. We believe our dataset has a good balance between representativeness (popular projects from different application domains) and overall execution time of the evaluation. The use of open-source projects also supports reproducibility and enables us to further confirm found crashes.

DS-CodaMosa: The new DS-C dataset targets evaluating modules using C-extensions. To evaluate our approach on an unbiased dataset, we additionally executed PYNGUIN with and without subprocess-execution on DS-CODAMOSA, previously used to evaluate CODAMOSA [22]. We decided to use this dataset as it is the largest one previously used for evaluating PYNGUIN with 486 modules from 27 projects.

2) Experiment Settings: We implemented subprocess-execution in PYNGUIN and executed it on each subject. We used a DOCKER container to isolate the executions from their environment based on Python 3.10.16. All tool runs were executed on dedicated servers, each equipped with an AMD EPYC 7443P CPU and 256 GB RAM. However, we only assigned a single CPU core and 4 GB of RAM to each run to simulate a more constrained environment, closer to the conditions in which the SUTs will have to operate, and to be able to save time by executing multiple runs in parallel.

We used PYNGUIN's implementation of DynaMOSA [51] with the same parameter settings and a search budget of 600 s, which was effective for test generation [21]. To minimise the influence of randomness we executed PYNGUIN on each subject 30 times with threaded, subprocess, heuristic, restart and combined mode each [61]. Assuming that all runs require the full budget of 600 s, we need 5 configurations $\times$ 1 648 modules $\times$ 600 s $\times$ 30 repetitions = 41 300 h to execute PYNGUIN with DS-C and 5 configurations $\times$ 486 modules $\times$ 600 s $\times$ 30 repetitions = 12 200 h to execute it with DS-CODAMOSA (not parallelised).

3) Experiment Procedure: As a basis for answering RQ1 and RQ2 we use the results of subprocess and threaded runs. We additionally use the results of heuristic, restart and combined runs when analyzing performance in RQ3. For every successful run, we collect the achieved branch coverage while all crashed runs achieve 0 % coverage, as crashed runs do not export any test cases. When using subprocess-execution, PYNGUIN exports a crash-revealing test case whenever it encounters a C-related crash during the execution of the module under test. With restart and combined mode, and when the first execution crashes, only the second execution attempt using subprocess-execution is considered for final coverage, as the first execution did not result in any test cases. The coverage during the first execution is set to 0 %. We use both, the collected coverage data and the generated crash-revealing tests to answer all research questions (RQ1, RQ2, RQ3).

To address RQ2, we need to ensure that all crash causes are reproducible. PYNGUIN's test execution, which uses the MULTIPROCESS module for performance reasons, can lead to crash-revealing tests not being generally reproducible. To resolve this, we re-executed all crash-revealing tests with PYTEST using new Python interpreters with the same version and dependencies. After re-execution, we removed non-reproducible tests, filtered the crash-revealing tests to only retain those causing timeouts or crashes when run in new interpreters, and recorded their exit codes. We chose to retain crash-revealing tests that consistently timed out after 30 s, as this duration is, in our judgment, sufficient to complete a typical test generated by PYNGUIN, and may indicate faults in the SUT. To obtain more precise information about the causes of the crashes in crash-revealing tests, we analysed the logs produced by Python's `faulthandler` module. These logs provide the traceback of the last Python function calls prior to the crashes, enabling us to heuristically group tests by the final Python function executed. We used a heuristic because, in Python 3.10, the `faulthandler` module only provides traceback dumps at the Python layer, but not at the C layer. Therefore, we know which Python functions were called immediately before the crashes, but we cannot determine whether they have a common or different cause at the C layer. Starting with Python 3.14, the `faulthandler` module also provides the C stack trace, but not all libraries in our dataset were compatible with this version, so we were unable to use it for our evaluation. After a manual analysis, we also decided to exclude some tests that contained calls to 3 types of functions that put the interpreter in an unstable state, such as the `__del__` functions. These were causing crashes when executing code that did not have any faults, thereby causing a bias. For further details, we refer to the replication package [23].

4) Evaluation Metrics: For each execution of a module, we keep track of the branch coverage over time as well as the overall branch coverage at the end of the search. To evaluate how many crashes subprocess-execution can avoid (RQ1), we count and compare the number of successful executions of PYNGUIN with threaded-execution and subprocess-execution. We measure PYNGUIN's ability to generate crash-revealing tests (RQ2) by counting the number of generated crash-revealing tests and checking how many of them are reproducible. To assess if one configuration performs better than another configuration in terms of avoided crashes (RQ1) and overall coverage (RQ3), we use the Mann-Whitney U-test [62] at $\alpha = 0.05$. We compute the Vargha and Delaney effect size $\hat{A}_{12}$ [63] to investigate the difference in the achieved overall coverage between two configurations. These metrics do not make assumptions on the data distribution and are in line with recommendations for assessing randomised algorithms [61] and were used in previous work [20], [15], [21], [22], [42].

B. Threats to Validity

Internal Validity. While the subprocess-execution (see Section III-B) isolates SUT crashes, the performance overhead of inter-process communication affects achieved coverage. Changes in performance may stem from the execution overhead rather than from crash isolation. Nevertheless, subprocess-execution enables testing of previously untestable modules, improving over threaded-execution. Furthermore, we added a heuristic which combines the benefits of both approaches.

Another threat is that some crashes found by generated crash-revealing test cases are not reproducible upon re-execution. Non-reproducible crashes may be falsely counted as faults, leading to an overestimation of fault-detection effectiveness. This behaviour is often due to non-determinism or the internal state of the SUT. Other factors might be the internal state of the execution environment or side effects introduced by PYNGUIN's instrumentation mechanisms. Such, so-called *flaky tests*, are a known issue for test generation tools [64] and do not specifically occur because of subprocess-execution.

External Validity. The generalisability of our findings may be limited. DS-C consists of 1 648 modules from 21 projects on PYPI, selected to include the most popular libraries with C-extensions and filtered using the GitHub API and specific criteria (e.g., the presence of C/C++ code). These design choices may introduce selection bias, potentially exclude projects or modules where subprocess-execution behaves differently, and limit the transferability of the results to other Python libraries or C-extension modules. The bugs identified are also tied to these libraries, and the prevalence of such bugs detectable by this method might vary. We mitigate this threat by evaluating PYNGUIN also on the DS-CODAMOSA dataset, which is unbiased and does not rely on the presence of C/C++ code.

Another possible threat regarding our DS-C dataset is that we only selected modules that are public according to Python, meaning their names do not start with an underscore. This approach removes modules that developers do not want to be used directly, and those that remain do not necessarily correspond to the public API defined in the documentation of the projects. There exists, however, no reliable way to automatically differentiate between public and private APIs in Python. We limited the dataset to public modules of projects, because the Python documentation for C-extension modules [29], [65] states that public functions defined in C must raise an exception in the case of a failure and not crash the interpreter. Thus, we can be sure that if a crash occurs, unless the implementation violates the Python documentation, there is either a bug in a public C-function or in the Python code calling a private C-function.

We do not compare PYNGUIN with other test generation tools, which introduces another threat to validity. However, to the best of our knowledge, there is no existing tool that can handle C-exceptions. Therefore, a comparison to other tools would not help to understand the improvements achieved by subprocess-execution. We instead focus on the comparison of PYNGUIN with and without subprocess-execution.

Construct Validity. The metrics used to evaluate the improvements focus on the crash avoidance of PYNGUIN (RQ1), the detection of crash causes (RQ2) and the impact on branch coverage (RQ3). When measuring branch coverage, we consider crashed runs to achieve 0 % coverage. This couples robustness gains with search effectiveness, which is intentional: crashed runs do not generate tests and therefore do not contribute to coverage. While these aspects are important indicators of bug-finding capability and robustness, they may not comprehensively capture all features of test case quality, such as the ability to detect non-crashing logical errors, the diversity of inputs, or the maintainability of the generated tests. However, we focus on crash-revealing tests, as they are the most relevant for our approach and leave the other aspects for future work. Furthermore, the evaluation does not deeply

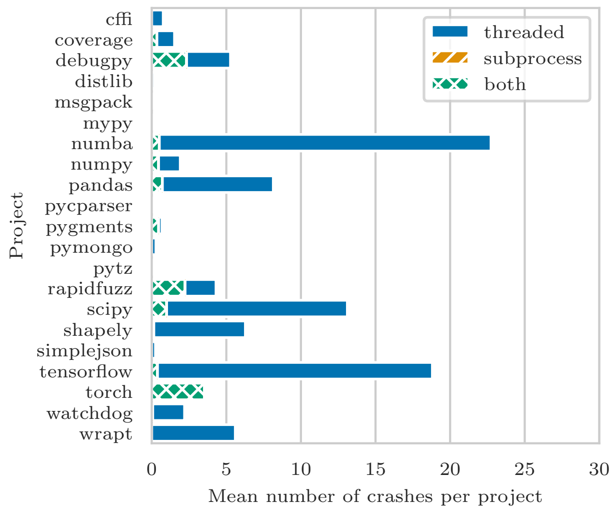


Figure 3: Distribution of crashes when using threaded-execution and subprocess-execution on DS-C.

analyse the severity or types of these crashes, which could provide a more nuanced understanding of the practical impact. Analyzing the severity is an important, yet a different problem, and thus we also leave it for future work. Finally, unique crash causes are identified using a heuristic approximation based on the last Python function called before the crash. Although this method is not perfect and may not accurately reflect the true native-level root causes of the crashes, it provides a first simple automated estimation. We acknowledge that this is a limitation of our approach and that a more accurate method using the `faulthandler` module of Python 3.14 could be used in future work to better classify the crash causes, once all SUTs support Python 3.14.

C. RQ1: Avoiding Crashes

Recall that, in the context of this paper, a crash refers to a fault that causes the Python interpreter to terminate unexpectedly, interrupting test generation, preventing coverage progress, and the discovery of additional faults. To determine how many crashes PYNGUIN can avoid by using subprocess-execution, we investigate the number of crashes that PYNGUIN encounters when using threaded-execution and subprocess-execution. Figure 3 shows the number of crashes averaged over all modules per project for PYNGUIN on DS-C. Table II additionally shows the number of modules where subprocess-execution had (significantly) less, equal, or (significantly) more crashes than threaded-execution and the Vargha and Delaney $\hat{A}_{12}$ effect size for an in-depth comparison.

Fewer Crashes with Subprocess-Execution. On projects that rely heavily on C/C++ extensions for performance-critical operations, such as the machine learning libraries NUMBA and TENSORFLOW, threaded-execution encounters many crashes while subprocess-execution avoids most of them. Subprocess-execution produced (significantly) fewer crashes than threaded-execution in 200 modules (sig: 184) and 624 modules (sig: 609) respectively. The mean $\hat{A}_{12}$ effect sizes of 0.874 and 0.821

Table II: Number of modules where subprocess-execution (treatment) had (significantly) fewer, equal, or (significantly) more crashes than threaded-execution (control) and the Vargha and Delaney $\hat{A}_{12}$ effect size for DS-C.

Project	Fewer (Sig)	Equal	More (Sig)	$\hat{A}_{12}$
cffi	2 (1)	11	0 (0)	0.513
coverage	6 (2)	30	2 (1)	0.520
debugpy	7 (4)	6	5 (1)	0.549
distlib	2 (0)	9	0 (0)	0.503
msgpack	0 (0)	1	0 (0)	0.500
mypy	0 (0)	4	0 (0)	0.500
numba	200 (184)	29	7 (0)	0.874
numpy	21 (7)	59	3 (0)	0.530
pandas	129 (87)	36	2 (0)	0.623
pycparser	0 (0)	10	0 (0)	0.500
pygments	5 (3)	105	5 (2)	0.504
pymongo	7 (2)	43	0 (0)	0.505
pytz	0 (0)	4	0 (0)	0.500
rapidfuzz	1 (1)	4	1 (1)	0.497
scipy	8 (5)	0	0 (0)	0.729
shapely	22 (13)	2	0 (0)	0.615
simplejson	1 (0)	2	0 (0)	0.506
tensorflow	624 (609)	113	8 (0)	0.821
torch	24 (14)	31	37 (26)	0.498
watchdog	2 (2)	12	1 (0)	0.536
wrapt	1 (1)	4	0 (0)	0.593
Total	**1062 (935)**	**515**	**71 (31)**	**0.717**

indicate a large positive effect. We conjecture this is because subprocess-execution prevents the interpreter from crashing by isolating the SUT in a separate process.

No/Few Execution Model Crashes. Some projects (e.g. the packaging library DISTLIB or the serialization library MSGPACK) show no crashes under either execution model. Others (e.g. the syntax highlighter PYGMENTS or the database library PYMONGO) exhibit very few crashes. These projects may be well-tested, exercise only pure-Python APIs, or PYNGUIN may not trigger the native-code paths that tend to cause interpreter-level failures. These projects have in common that they are primarily written in pure Python and use less C code.

Both Execution Models Crash. Some crashes occur regardless of the execution mode. Those indicate limitations inherent to the PYNGUIN tool, such as its inability to handle coroutines, and that subprocess-execution does not address. For example, projects like the automated release tool PYTHON-SEMANTIC-RELEASE or the project scaffolding tool COOKIECUTTER require specific environmental conditions to run correctly, such as the presence of a version-controlled repository, specific configuration files, or network connectivity. PYNGUIN is currently not able to emulate these conditions, leading to crashes regardless of the isolation mode used. As these are not related to the execution isolation provided by subprocess-execution, we pose addressing them as future work.

More Crashes with Subprocess-Execution. For a few modules, primarily within the PYTORCH library, subprocess-execution shows more crashes than threaded-execution. Manual inspection indicates these are due to infrastructure issues: we allocated a total of 10 h for 30 repetitions for each module to account 10 min for dependency installation and 10 min for test generation. Installing PYTORCH includes large dependencies such as

Table III: Distribution of fault types among unique crash causes.

Crash reason	Exit code	Signal	Count	Percentage
Aborted	−6	SIGABRT	25	11.7 %
Segmentation fault	−11	SIGSEGV	185	86.9 %
Timeout	None	N/A	3	1.41 %

CUDA binaries which took more than 10 min on average. This lead to container timeouts depending on the current infrastructure load and affected both execution models. These cases do not indicate a flaw in subprocess-execution but are artefacts of the execution environment.

Overall, the results show an improvement with subprocess-execution, which increased the number of successful, crash-free modules from 625 to 1 438. This corresponds to a 56.5 % relative reduction in execution failures. The superior isolation of subprocess-execution was especially evident with 215 modules, for which PYNGUIN achieved at least one successful run, while it consistently failed to generate test cases for all 30 runs when using threaded-execution. The mean Vargha and Delaney effect sizes $\hat{A}_{12}$ of 0.717 indicates a large positive effect of subprocess-execution over threaded-execution.

Summary (RQ1: Avoiding Crashes)

We can avoid up to 56.5 % of all crashes by using subprocess-execution. The exact number of avoided crashes depends on the project.

D. RQ2: Unique Crash Causes

As shown in RQ1, subprocess-execution can prevent Pynguin's process from crashing, but it can also store the tests that would have caused the crashes to help developers reproduce them. We call those crash-revealing tests, and PYNGUIN generated 120 176 of them. However, some tests no longer caused crashes when re-executed, which decreased the total of reproducible tests to 114 711. After grouping them using their last Python instructions called before crashing and their exit codes, we found 213 unique crash causes. Based on this, we identified 32 previously unknown faults and reported them to the respective development teams. Most of these faults have been confirmed by the developers and included in the projects' development processes, while others have not yet been reviewed or have been classified as "won't fix." Due to a lack of time and resources from the developers, only a few have been fixed so far. Since quantitative data does not provide any insights into the severity of the detected faults, we classified the crashes based on their exit codes, as shown in Table III. PYNGUIN supports a subset of all POSIX signal exit codes [66] that may be raised due to programming errors: *segmentation faults*, *aborteds*, *illegal instructions*, *bus errors*, *out of memory (OOM) kills*, and *floating-point exceptions*.

Segmentation Faults are Predominant. *Segmentation faults* occur when the SUT attempts to access memory outside its allocated range, causing the OS to terminate the process. Since C provides no built-in safeguards against such behaviour, we expected *segmentation faults* to be the most common crash type

```
import tensorflow.python.eager.context as module_0

def test_case_0():
    none_type_0 = None
    var_0 = module_0.initialize_logical_devices()
```

Figure 4: Invoking `initialize_logical_devices` causes a *segmentation fault*.

```
import tensorflow.python.eager.tape as module_0


def test_case_0():
    list_0 = []
    tape_0 = module_0.Tape(list_0)
    module_0.push_tape(tape_0)
    module_0.variable_accessed(tape_0)
```

Figure 5: Invoking `variable_accessed` with an invalid argument and after setting up the internal state of the SUT causes a *segmentation fault*.

and we found 185 of them. We also detected 25 *aborteds* which usually stem from C assertions. Developers use such assertions to catch bugs as early as possible, by checking preconditions or invariants. Since they are a faily widespread practice, this is expected. These *aborteds* should, however, never reach the Python level because developers are expected to raise Python exceptions instead of letting the interpreter crash, as discussed in Section IV-B. Finally, we observed 3 *timeouts*, as defined in Section IV-A3. We tried to re-execute them for a longer time, but they still did not terminate. They may indicate an infinite loop or a deadlock in the SUT and are thus noteworthy.

Automated State-Dependent Fault Detection. In certain cases, such as in Fig. 4, fuzzing might find the same bugs as our approach by calling the function with random arguments. In this case, a *segmentation fault* is raised from a public module if the computer does not have a tensor processing unit, TPU. Nonetheless, PYNGUIN is able to automatically detect such straightforward faults as well as more complex cases that otherwise require domain knowledge and manually configuring fuzzers. For example, in Fig. 5, the internal state of the SUT is first initialised and then the `variable_accessed` function is called with an invalid argument, causing a crash. Although fuzzers can invoke the function, the crash occurs only in a special SUT state. This requires manually configuring the fuzzer. In contrast, PYNGUIN reveals these crashes fully automatically requiring neither configuration nor domain knowledge.

Non-Reproducible Crash-Revealing Tests. 5 465 of 120 176 detected crashes could not be reproduced when executing the exported test cases. It is unlikely that there was no crash in the first place or else PYNGUIN would not have exported them. Section IV-B explains several reasons for this behaviour. The most noteworthy case is that we had to remove the few *OOM kills* because they mainly came from NUMPY, which was trying to allocate arrays of dozens of GB in the generated crash-revealing tests, inevitably causing false positives.

Summary (RQ2: Unique Crash Causes)

Using subprocess-execution, we detected 213 unique crash causes and identified 32 unknown faults. The crashes were predominantly *segmentation faults* (86.9 %), with smaller proportions of *aborteds* from C assertion violations (11.7 %) and *timeouts* (1.41 %).

E. RQ3: Branch Coverage

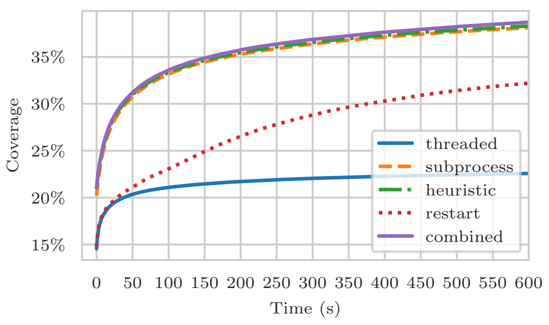
Figure 6 and Table IV show the performance differences between threaded, subprocess, heuristic, restart and combined mode for DS-C and DS-CODAMOSA.

Higher Coverage with Subprocess-Execution. On DS-C, PYNGUIN with subprocess-execution achieves a higher branch coverage than with threaded-execution (Fig. 6a). When using the heuristic to automatically select the execution model and when combining this with restart (combined), PYNGUIN achieves an even higher branch coverage. Solely using restart results in coverage similar to threaded-execution initially and later improving towards subprocess-execution. Our observations hold for the overall branch coverage, and these differences are statistically significant, as Table IVa shows. On 1 072 (987 significant) modules, subprocess-execution achieves a (significantly) higher branch coverage than with threaded-execution, while on 331 (247 significant) modules threaded-execution achieves a (significantly) higher branch coverage than with subprocess-execution. The mean Vargha and Delaney $\hat{A}_{12}$ effect size confirms a medium positive effect of 0.693. Using the heuristic to select the execution model and combining it with restart (combined) further increases the average performance.

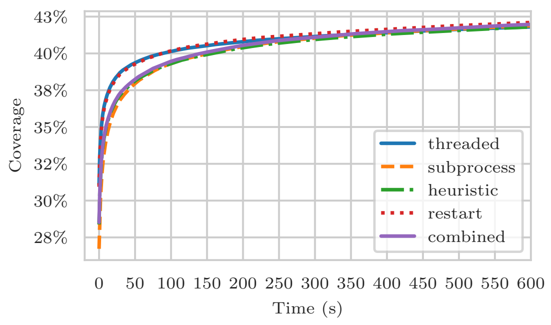
Overall, subprocess-execution is better than threaded-execution and combined performs best on DS-C. However, as this increase is heavily influenced by those cases where PYNGUIN with threaded-execution crashes and thus results in 0 % coverage, we also investigate how branch coverage results generalize on DS-CODAMOSA.

Lower Coverage with Subprocess-Execution. Even though Fig. 6b suggests that PYNGUIN with threaded-execution achieves similar coverage to subprocess-execution on DS-CODAMOSA, Table IVb shows that PYNGUIN with threaded-execution performs better than with subprocess-execution on most modules in DS-CODAMOSA. On 53 (37 significant) modules, subprocess-execution achieves significantly higher branch coverage than with threaded-execution, while on 199 (161 significant) modules threaded-execution achieves (significantly) higher branch coverage than with subprocess-execution. The mean Vargha and Delaney $\hat{A}_{12}$ effect size confirms a small negative effect of 0.418. Overall, threaded-execution is better than subprocess-execution and restart mode performs best on DS-CODAMOSA.

Strategy to Use. The best test execution strategy depends on whether the SUT uses FFI modules that might trigger C-related errors. As this is generally unknown in advance, we implemented three approaches to automatically decide which execution mode to use (see Section III-B2). The heuristic selects subprocess-execution for 73.9 % of DS-C modules and



(a) Branch coverage over time on DS-C.



(b) Branch coverage over time on DS-CodaMosa.

Figure 6: Branch coverage over time when running Pynguin.

(a) Number of modules and effect size for DS-C.

Treatment	Better (Sig)	Equal	Worse (Sig)	$\hat{A}_{12}$
combined	1109 (986)	258	281 (134)	0.716
heuristic	1111 (976)	254	283 (147)	0.713
subprocess	1072 (987)	245	331 (247)	0.693
restart	1075 (675)	325	248 (34)	0.629
threaded	0 (0)	1648	0 (0)	0.500

(b) Number of modules and effect size for DS-CodaMosa.

Treatment	Better (Sig)	Equal	Worse (Sig)	$\hat{A}_{12}$
restart	74 (10)	349	63 (1)	0.504
threaded	0 (0)	486	0 (0)	0.500
combined	60 (30)	266	160 (112)	0.445
heuristic	48 (24)	270	168 (111)	0.442
subprocess	53 (37)	234	199 (161)	0.418

Table IV: Number of modules where each treatment configuration (Treatment) performed (significantly) better, equal or (significantly) worse than threaded-execution (control) and the Vargha and Delaney $\hat{A}_{12}$ effect size.

63.3 % of DS-CodaMosa modules. The high rate on DS-CodaMosa stems from the heuristic's conservative design, which chooses subprocess-execution for any module where C-extensions are imported, regardless of usage or potential to crash. Restart was triggered on 59.1 % of DS-C modules and on 11.1 % of DS-CodaMosa modules. Table IVa shows that the combined mode performs best on DS-C, achieving a large mean Vargha and Delaney $\hat{A}_{12}$ effect size of 0.716 compared to threaded-execution. On DS-CodaMosa, Table IVb shows that only the restart mode performs better than threaded-execution, achieving a negligible mean Vargha and Delaney $\hat{A}_{12}$ effect size of 0.504. Overall, we recommend using restart mode, as it performs better than threaded-execution on both datasets and adds the stability of subprocess-execution only when needed.

Summary (RQ3: Branch Coverage)

Subprocess-execution achieves lower branch coverage than threaded-execution in general. However, on modules that use C-extensions, subprocess-execution outperforms threaded-execution. Restarting Pynguin with subprocess-execution only when threaded-execution crashes (restart) yields the best performance trade-off.

V. Related Work

Automated fault detection typically relies on crash or exception oracles, common in fuzzing [19] and SBSE tools like EvoSuite [16], [17]. However, unlike crash reproduction techniques that require existing stack traces [67], [68], [69], [70], our approach discovers and generates reproducible crash-revealing tests from scratch using only the SUT. Furthermore, Pynguin serves as a general-purpose tool, avoiding the domain-specific constraints, such as used for RESTful testing [71] or the static analysis dependencies of API misuse detection [72]. Finally, while most prior research targets Java or focuses on static analysis of Python C-extensions [10], to the best of our knowledge, this is the first large-scale study targeting dynamic fault detection for Python C-extensions.

VI. Conclusions and Future Work

We introduced subprocess-execution to automatically generate crash-revealing tests for Python modules that use C-extensions. By executing the SUT in a separate process, subprocess-execution captures crashes, generates crash-revealing tests and allows the test generation to continue after a crash caused by a C-extension module. Our evaluation on 1 648 modules from 21 libraries shows that our approach avoids up to 56.5 % of crashes during test generation. It produced 114 711 crash-revealing tests, allowing us to identify 32 unknown faults.

A large proportion of the bugs found with our approach stem from missing validation of arguments passed to C functions, which can cause crashes in C due to unmet assumptions. To address this, future work should measure code coverage not only in the Python layer but also in the underlying FFI code, and enhance Pynguin's fitness function with this information.

Additionally, future work can leverage subprocess-execution to implement parallel test execution. This would address one of the bottleneck of test generation—fitness evaluation—and may overcome the performance overhead of subprocess-execution.

Finally, while our approach focuses on testing the Python projects that use C-extensions, the underlying idea of executing the SUT in a separate process is not limited to Python. It can be applied to any programming language with process isolation.

VII. Acknowledgment

This research was partially funded by the CyberExcellence by DigitalWallonia project (No. 2110186) funded by the Public Service of Wallonia (SPW Recherche), and with the support of Wallonie-Bruxelles International (RealM project), as well as by the German Research Foundation (DFG) under grant FR 2955/5-1 (TYPES4STRINGS: Types For Strings).

References

[1] Stack Overflow. (2023) Stack Overflow Developer Survey 2023: Most popular technologies. Accessed: 2025-07-17. [Online]. Available: https://survey.stackoverflow.co/2023/#most-popular-technologies-language-prof

[2] NumPy Developers. (2025) NumPy. Accessed: 2025-07-01. [Online]. Available: https://numpy.org/

[3] Pandas Developers. (2025) Pandas. Accessed: 2025-07-01. [Online]. Available: https://pandas.pydata.org/

[4] TensorFlow Developers. (2025) TensorFlow. Accessed: 2025-07-01. [Online]. Available: https://www.tensorflow.org/

[5] SciPy Developers. (2025) SciPy. Accessed: 2025-07-12. [Online]. Available: https://scipy.org/

[6] Cython Developers. (2025) Cython: C-Extensions for Python. Accessed: 2025-07-01. [Online]. Available: https://cython.org/

[7] ——. (2025) Cython User guide: Compiler directives. Accessed: 2025-07-01. [Online]. Available: https://cython.readthedocs.io/en/3.1.x/src/userguide/source_files_and_compilation.html#compiler-directives

[8] N. Mitchell and G. Sevitsky, "LeakBot: An automated and lightweight tool for diagnosing memory leaks in large java applications," in *Proc. ECOOP*, ser. LNCS, vol. 2743. Springer, 2003, pp. 351–377.

[9] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proc. PLDI*. ACM, 2007, pp. 89–100.

[10] M. Hu and Y. Zhang, "The Python/C API: Evolution, usage statistics, and bug patterns," in *Proc. SANER*, 2020, pp. 532–536.

[11] B. Yu, C. Tian, N. Zhang, Z. Duan, and H. Du, "A dynamic approach to detecting, eliminating and fixing memory leaks," *J. Comb. Optim.*, vol. 42, no. 3, pp. 409–426, 2021.

[12] Bloomberg. (2025) Memray. Accessed: 2025-06-16. [Online]. Available: https://github.com/bloomberg/memray

[13] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for Java," in *Proc. OOPSLA Companion*. ACM, 2007, pp. 815–816.

[14] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *Proc. ESEC/FSE*. ACM, 2011, pp. 416–419.

[15] S. Lukasczyk and G. Fraser, "Pynguin: Automated unit test generation for Python," in *Proc. ICSE Companion*, 2022, pp. 168–172.

[16] G. Fraser and A. Arcuri, "EvoSuite: On the challenges of test case generation in the real world," in *Proc. ICST*, 2013, pp. 362–369.

[17] ——, "1600 faults in 100 projects: automatically finding faults while achieving high coverage with EvoSuite," *Empir. Softw. Eng.*, vol. 20, no. 3, pp. 611–639, 2015.

[18] pytest Developers. (2025) pytest Documentation. Accessed: 2025-06-27. [Online]. Available: https://docs.pytest.org/en/stable/

[19] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A Survey for Roadmap," *ACM Comput. Surv.*, vol. 54, no. 11, pp. 230:1–230:36, 2022.

[20] S. Lukasczyk, F. Kroiß, and G. Fraser, "Automated unit test generation for Python," in *Proc. SSBSE*, ser. LNCS, vol. 12420. Springer, 2020, pp. 9–24.

[21] ——, "An empirical study of automated unit test generation for Python," *Empir. Softw. Eng.*, vol. 28, no. 2, p. 36, 2023.

[22] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *Proc. ICSE*, 2023, pp. 919–931.

[23] L. Berg, L. Krodinger, S. Lukasczyk, A. Panichella, G. Fraser, W. Vanhoof, and X. Devroey, "Artefact for the paper "Real-World Fault Detection for C-Extended Python Projects with Automated Unit Test Generation" ," Mar. 2026. [Online]. Available: https://doi.org/10.5281/zenodo.18879486

[24] Oracle. (2025) Java Native Interface (JNI). Accessed: 2025-07-01. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/jni/

[25] Java Native Access (JNA) Contributors. (2025) Java Native Access (JNA). Accessed: 2025-07-01. [Online]. Available: https://github.com/java-native-access/jna

[26] HaskellWiki contributors. (2025) Foreign Function Interface. Accessed: 2025-07-01. [Online]. Available: https://wiki.haskell.org/Foreign_Function_Interface

[27] Python Software Foundation. (2025) ctypes — A foreign function library for Python. Accessed: 2025-07-01. [Online]. Available: https://docs.python.org/3/library/ctypes.html

[28] cffi developers. (2025) CFFI Documentation (Stable). Accessed: 2025-07-01. [Online]. Available: https://cffi.readthedocs.io/en/stable/

[29] Python Software Foundation. (2025) Extending Python with C or C++. Accessed: 2025-07-01. [Online]. Available: https://docs.python.org/3/extending/extending.html

[30] SWIG Developers. (2025) SWIG: Simplified Wrapper and Interface Generator. Accessed: 2025-07-01. [Online]. Available: https://www.swig.org/

[31] Python Software Foundation. (2025) faulthandler — Dump the Python traceback. Accessed: 2025-07-01. [Online]. Available: https://docs.python.org/3/library/faulthandler.html

[32] G. Fraser and J. M. Rojas, "Software testing," in *Handbook of Software Engineering*. Springer, 2019, pp. 123–192.

[33] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proc. ICSE*. IEEE Comp. Soc., 2007, pp. 75–84.

[34] P. Tonella, "Evolutionary testing of classes," in *Proc. ISSTA*. ACM, 2004, pp. 119–128.

[35] J. Campos, Y. Ge, N. Albunian, G. Fraser, M. Eler, and A. Arcuri, "An empirical evaluation of evolutionary algorithms for unit test suite generation," *Inf. Softw. Technol.*, vol. 104, pp. 207–235, 2018.

[36] A. Panichella, F. M. Kifetew, and P. Tonella, "A large scale empirical comparison of state-of-the-art search-based test case generators," *Inf. Softw. Technol.*, vol. 104, pp. 236–256, 2018.

[37] S. Dola, M. B. Dwyer, and M. L. Soffa, "Distribution-aware testing of neural networks using generative models," in *Proc. ICSE*. IEEE, 2021, pp. 226–237.

[38] A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais, "Effective test generation using pre-trained Large Language Models and mutation testing," *Inf. Softw. Technol.*, vol. 171, p. 107468, 2024.

[39] C. Yang, J. Chen, B. Lin, J. Zhou, and Z. Wang, "Enhancing LLM-based test generation for hard-to-cover branches via program analysis," *CoRR*, vol. abs/2404.04966, 2024.

[40] J. A. Pizzorno and E. D. Berger, "CoverUp: Effective high coverage test generation for python," *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, pp. 2897–2919, 2025.

[41] G. Ryan, S. Jain, M. Shang, S. Wang, X. Ma, M. K. Ramanathan, and B. Ray, "Code-aware prompting: A study of coverage-guided test generation in regression setting," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 951–971, 2024.

[42] R. Yang, X. Xu, and R. Wang, "LLM-enhanced evolutionary test generation for untyped languages," *Autom. Softw. Eng.*, vol. 32, no. 1, p. 20, 2025.

[43] D. Xiao, Y. Guo, Y. Li, and L. Chen, "Optimizing search-based unit test generation with large language models: An empirical study," in *Proc. Internetware*. ACM, 2024, pp. 71–80.

[44] K. Jain and C. L. Goues, "TestForge: Feedback-driven, agentic test suite generation," *CoRR*, vol. abs/2503.14713, 2025.

[45] A. Sapozhnikov, M. Olsthoorn, A. Panichella, V. Kovalenko, and P. Derakhshanfar, "TestSpark: IntelliJ IDEA's ultimate test generation companion," in *Proc. ICSE Companion*. ACM, 2024, pp. 30–34.

[46] A. Abdullin, P. Derakhshanfar, and A. Panichella, "Test wars: A comparative study of SBST, symbolic execution, and LLM-based approaches to unit test generation," in *Proc. ICST*. IEEE, 2025, pp. 221–232.

[47] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, "ChatUniTest: A framework for LLM-based test generation," in *Proc. FSE Companion*. ACM, 2024, pp. 572–576.

[48] W. C. Ouedraogo, K. Kabore, H. Tian, Y. Song, A. Koyuncu, J. Klein, D. Lo, and T. F. Bissyande, "LLMs and prompting for unit test generation: A large-scale evaluation," in *Proc. ASE*. ACM, 2024, pp. 2464–2465.

[49] S. Roychowdhury, G. Sridhara, A. K. Raghavan, J. Bose, S. Mazumdar, H. Singh, S. B. Sugumaran, and R. Britto, "Static program analysis guided LLM based unit test generation," *CoRR*, vol. abs/2503.05394, 2025.

[50] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Trans. Software Eng.*, vol. 50, no. 1, pp. 85–105, 2024.

[51] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Trans. Software Eng.*, vol. 44, no. 2, pp. 122–158, 2018.

[52] UQFoundation. (2025) dill. Accessed: 2025-06-16. [Online]. Available: https://github.com/uqfoundation/dill

[53] M. McKerns and M. Aivazis, "pathos: a framework for heterogeneous computing," 2010-. [Online]. Available: https://uqfoundation.github.io/project/pathos

[54] M. M. McKerns, L. Strand, T. Sullivan, A. Fang, and M. A. G. Aivazis, "Building a framework for predictive science," *CoRR*, vol. abs/1202.1056, 2011.

[55] UQFoundation. (2025) pathos: multiprocess module. Accessed: 2025-06-16. [Online]. Available: https://github.com/uqfoundation/pathos

[56] Python Software Foundation. (2025) inspect.getsource — Python Docs. Accessed: 2025-07-15. [Online]. Available: https://docs.python.org/3/library/inspect.html#inspect.getsource

[57] AlDanial. (2025) cloc: Count Lines of Code. Accessed: 2025-07-09. [Online]. Available: https://github.com/AlDanial/cloc

[58] GitHub. (2025) GitHub REST API: List repository languages. Accessed: 2025-07-09. [Online]. Available: https://docs.github.com/en/rest/repos/repos?apiVersion=2022-11-28#list-repository-languages

[59] B. Taskaya, "Reiz: Structural source code search," *J. Open Source Softw.*, vol. 6, no. 62, p. 3296, 2021.

[60] Hugo van Kemenade. (2025) Top PyPI Packages. Accessed: 2025-05-12. [Online]. Available: https://hugovk.github.io/top-pypi-packages/

[61] A. Arcuri and L. C. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Softw. Test. Verification Reliab.*, vol. 24, no. 3, pp. 219–250, 2014.

[62] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 1947, publisher: Institute of Mathematical Statistics.

[63] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *J. Educ. Behav. Stat.*, vol. 25, no. 2, pp. 101–132, 2000.

[64] M. Gruber, M. F. Roslan, O. Parry, F. Scharnböck, P. McMinn, and G. Fraser, "Do automatc test generation tools generate flaky tests?" in *Proc. ICSE*. ACM, 2024, pp. 1–12.

[65] Python Software Foundation. (2025) Intermezzo: Errors and Exceptions. Accessed: 2025-07-01. [Online]. Available: https://docs.python.org/3/extending/extending.html#intermezzo-errors-and-exceptions

[66] Michael Kerrisk. (2025) signal(7) - Linux manual page. Accessed: 2025-07-18. [Online]. Available: https://man7.org/linux/man-pages/man7/signal.7.html

[67] J. Rößler, A. Zeller, G. Fraser, C. Zamfir, and G. Candea, "Reconstructing core dumps," in *Proc. ICST*. IEEE, 2013, pp. 114–123.

[68] M. Soltani, A. Panichella, and A. van Deursen, "A guided genetic algorithm for automated crash reproduction," in *Proc. ICSE*. IEEE, 2017, pp. 209–220.

[69] P. Derakhshanfar, X. Devroey, A. Panichella, A. Zaidman, and A. van Deursen, "Botsing, a search-based crash reproduction framework for Java," in *Proc. ASE*, 2021, pp. 1278–1282.

[70] A. Bergel and I. S. Munoz, "Beacon: Automated Test Generation for Stack-Trace Reproduction using Genetic Algorithms," in *Proc. SBSE@ICSE*. IEEE, 2021.

[71] A. Arcuri, "RESTful API automated test case generation with EvoMaster," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 1, pp. 3:1–3:37, 2019.

[72] M. Kechagia, X. Devroey, A. Panichella, G. Gousios, and A. van Deursen, "Effective and efficient API misuse detection via exception propagation and search-based testing," in *Proc. ISSTA*. ACM, 2019, pp. 192–203.