On the Implementation of OS-Specific Tests: The CPython Case

Ricardo Job
UNINFO, IFPB
Cajazeiras, Brazil
ricardo.job@ifpb.edu.br

Andre Hora
Department of Computer Science, UFMG
Belo Horizonte, Brazil
andrehora@dcc.ufmg.br

ABSTRACT

Modern software systems are frequently developed and tested across multiple platforms (*e.g.*, Windows, Linux, and macOS). In the software testing context, practitioners adapt the tests to run differently according to the target platform. These tests, which need to identify the platform on which they will be executed, are referred to as OS-specific tests. In this paper, we present an empirical study to evaluate how developers implement OS-specific tests in CPython, which is the reference implementation project for the Python programming language. Then, we mine this project and assess their OS-specific tests quantitatively. For this, we propose three research questions to assess the frequency, location, and issues related to OS-specific tests. Our results show that OS-specific tests are common in the CPython project, and 13% of the analyzed test files are OS-specific tests (RQ1). OS Identification APIs are used more frequently in test code (53.46%), and the test decorator `@unittest.skipUnless` is the most used to skip tests depending on the platform (RQ2). We also find 170 issues related to OS-specific tests in CPython, and Windows is the most targeted platform (RQ3). Lastly, we discussed practical implications for practitioners and researchers. Based on our findings, we emphasized the importance of testing across multiple platforms and examined the relationship between issues and OS-specific tests, among other insights.

KEYWORDS

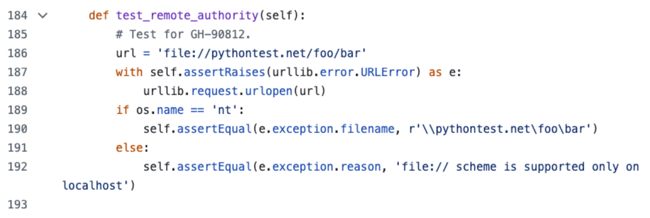
software testing, mining software repositories, test smells, Python

1 Introduction

Modern software systems are frequently developed and tested across multiple platforms. Indeed, modern CI/CD tools facilitate development and testing across Windows, Linux, and macOS [9]. Consequently, practitioners may need to handle *platform-specific code* in production or test code. In this context, a platform-specific code refers to code that adapts its behavior according to the platform on which it is executed. For instance, Windows and Unix-based systems have distinct file path conventions, requiring code adaptation to ensure correct execution across multiple platforms.

In the software testing context, practitioners may adapt the tests to run differently according to the target platform. These tests, which need to identify the platform on which they will be executed, are referred to as *OS-specific tests* [15]. OS-specific tests may make the test suite more flexible [22] and can change test behavior according to the executed platform. While such tests provide execution flexibility, they may also be associated with test smells. For instance, when the execution platform is identified through a conditional check, certain test branches or assertions may remain unexecuted, leading to conditional test logic and potentially resulting in rotten green tests [1, 7, 27, 28].

In practice, the OS-specific tests may execute different lines of code within the test code depending on the platform on which they are run. Figure 1 shows an example of an OS-specific test extracted from the CPython project, which is the canonical implementation of the Python programming language [10]. The test function `test_remote_authority`[1] uses the API `os.name`[2] to identify the current platform (line 189). The `if` block verifies whether the platform is Windows and performs Windows-specific assertions, while the `else` block checks for other platforms, such as macOS and Linux.



```
184     def test_remote_authority(self):
185         # Test for GH-90812.
186         url = 'file://pythontest.net/foo/bar'
187         with self.assertRaises(urllib.error.URLError) as e:
188             urllib.request.urlopen(url)
189         if os.name == 'nt':
190             self.assertEqual(e.exception.filename, r'\\pythontest.net\foo\bar')
191         else:
192             self.assertEqual(e.exception.reason, 'file:// scheme is supported only on
    localhost')
193
```

Figure 1: OS-specific test in the CPython project (`test_urllib`).

Indeed, OS-specific tests are common in CPython: we identified over 150 test files with OS-specific tests, similar to Figure 1. Despite being commonly used, we still do not know *how* developers implement these tests in the CPython repository. This knowledge can be used to understand OS-specific tests better and discover common issues faced by practitioners when testing across multiple platforms.

In this paper, we provide an empirical study to evaluate OS-specific tests in the CPython repository. Specifically, we mine this project and assess its OS-specific tests quantitatively. We propose three research questions to support our study:

- *RQ1: How frequent are OS-specific tests in the CPython project?* OS-specific tests are common in the CPython project. We find that 13% of the analyzed test files are OS-specific tests. `sys.platform` is the most used OS identification API, while Windows is the most targeted platform.
- *RQ2: Where are OS identification APIs implemented in tests?* OS Identification APIs in tests are used more frequently in test code (53.46%) than in test decorators (46.54%). The decorator `@unittest.skipUnless` is the most used to skip tests depending on the platform.
- *RQ3: What issues are related to OS-specific tests?* We find 170 issues related to OS-specific tests in CPython. 22.89% of the

[1]https://github.com/python/cpython/blob/bbe9c31edc4fc3e1cdc908e9a06593c394f4bfdb/Lib/test/test_urllib.py#L189
[2]https://docs.python.org/3.11/library/os.html#os.name

OS identification APIs have associated issues, and Windows is the most targeted platform (47%).

Finally, we discuss practical implications for practitioners and researchers. Based on our findings, we highlight the importance of testing on multiple platforms. We discuss the relationship between issues and related to OS-specific tests. For example, we find 170 issues related to OS-specific tests, and 27.06% of the test decorators and 19.25% of the test code are directly associated with at least one issue. We discuss problems in OS-specific test code that may propagate to CPython-dependent projects, leading to cascading failures. Lastly, we relate some of these issues to potential vulnerabilities, underlining the security relevance of this testing practice.

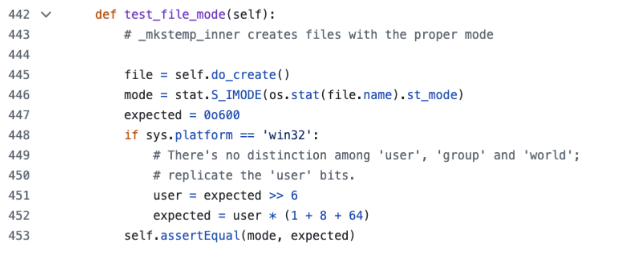
Contributions: The contributions of this paper are twofold: (1) we provide an empirical study to explore the usage of OS-specific tests, and (2) we propose actionable implications for researchers and practitioners concerned with cross-platform testing.

Structure: Section 2 reviews background and related work. Section 3 describes the study design, and Section 4 presents the results. Section 5 discusses the implications, while Section 6 outlines threats to validity. Finally, Section 7 concludes the paper.

2 Background and Related Work

2.1 OS Identification APIs

An *OS identification API* determines the platform (*e.g.,* operating system) on which the code is running. Developers often rely on these APIs to address platform-specific requirements [14]. For example, operating systems have distinct path name conventions, particularly between Windows and Unix-style paths, necessitating platform-aware adaptations in the code. Figure 2 illustrates a test (`test_tempfile`) that exemplifies this need. The test function `test_file_mode`[3] uses the OS Identification API `sys.platform`[4] (line 448) to determine the current platform and ensure that the permissions are handled correctly across Windows. The `if` block contains Windows-specific adaptations, allowing the developer to apply platform-specific logic within the test function when running on Windows.



```
442      def test_file_mode(self):
443          # _mkstemp_inner creates files with the proper mode
444
445          file = self.do_create()
446          mode = stat.S_IMODE(os.stat(file.name).st_mode)
447          expected = 0o600
448          if sys.platform == 'win32':
449              # There's no distinction among 'user', 'group' and 'world';
450              # replicate the 'user' bits.
451              user = expected >> 6
452              expected = user * (1 + 8 + 64)
453          self.assertEqual(mode, expected)
```

Figure 2: Example usage of OS identification API `sys.platform` (`test_tempfile`).

In Python, the usage of OS identification API is common in test code, and several APIs support checking the current operating system [15]. The Python Standard Library provides multiple OS-related interfaces. Three relevant libraries offer OS identification functionalities: `sys`,[5] `os`,[6] and `platform`.[7] These libraries expose various platform and OS-related functions. For instance, the API `sys.platform`[8] is explicitly described as: *"This string contains a platform identifier"*. Collectively, these libraries expose 15 APIs for OS identification, as summarized in Table 1. Understanding these APIs is essential to identifying how they are used in tests across platforms. Notably, ten APIs belong to the `platform` library, three to `os`, and two to `sys`.

Table 1: Common OS Identification APIs.

API	Short Description
`sys.platform`	Returns a platform identifier.
`sys.getwindowsversion`	Returns the current Windows version.
`os.name`	Name of the OS dependent module imported.
`os.supports_bytes_environ`	True if the native OS type of the environment is bytes.
`os.uname`	Returns information identifying the current OS.
`platform.platform`	Returns a detailed string identifying the underlying platform.
`platform.system`	Returns the system/OS name.
`platform.version`	Returns the system's release version
`platform.uname`	Returns the platform attributes.
`platform.win32_edition`	Get additional version information from the Windows Registry.
`platform.win32_ver`	Returns a string representing the current Windows edition.
`platform.win32_is_iot`	Return True if the Windows edition returned by `win32_edition` is recognized as an IoT edition.
`platform.mac_ver`	Get macOS version information and return it as a tuple
`platform.libc_ver`	Attempts to determine the libc version linked to the executable.
`platform.freedesktop_os_release`	Extracts OS identification from os-release and returns it as a dictionary.

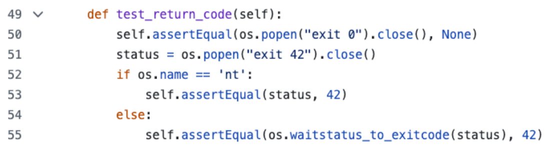
2.2 OS-Specific Tests

Section 2.1 introduced the concept of OS identification APIs, which allow developers to determine the platform on which their code is running. Consequently, tests must often be adapted to execute differently depending on whether the OS is Windows or Unix.

In this context, an *OS-specific test* is a test that identifies the platform on which it will be executed. That is, a test that calls an OS identification API. These tests may execute different lines of application code depending on the OS, making the test suite more flexible [22]. However, they can also alter test behavior based on the execution environment. Developers typically use OS identification APIs in tests either (1) directly within test code or (2) in test decorators. While inline platform checks enable conditional assertions and behavior, decorators offer a cleaner syntax to enable or disable entire tests based on the detected platform.

[3]https://github.com/python/cpython/blob/bbe9c31edc4fc3e1cdc908e9a06593c394f4bfdb/Lib/test/test_tempfile.py#L442-L453
[4]https://docs.python.org/3.11/library/sys.html#sys.platform
[5]https://docs.python.org/3.11/library/sys.html
[6]https://docs.python.org/3.11/library/os.html
[7]https://docs.python.org/3.11/library/platform.html
[8]https://docs.python.org/3.11/library/sys.html#sys.platform

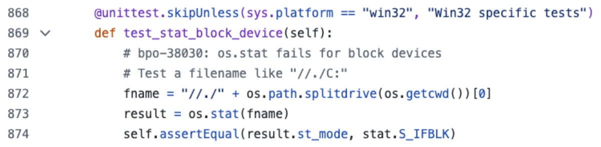
Test code. Figure 3 presents an example where an OS identification API is used within test code. The test function `test_return_code` calls on the API `os.name` (line 52) to adjust expected result based on the platform.[9] Note that OS identification APIs may appear in both test methods and test support methods.



```
49    def test_return_code(self):
50        self.assertEqual(os.popen("exit 0").close(), None)
51        status = os.popen("exit 42").close()
52        if os.name == 'nt':
53            self.assertEqual(status, 42)
54        else:
55            self.assertEqual(os.waitstatus_to_exitcode(status), 42)
```

Figure 3: Example OS-specific test in test method. The OS identification API `os.name` occurs directly in the test code.

Test decorators. Another approach is to use the OS Identification test decorators. The two major Python testing frameworks (unittest [35] and pytest [30]) provide decorators for conditionally skipping tests based on specific conditions [2]. Figure 4 illustrates an example in which the OS Identification API `sys.platform` is used in the test decorator `@unittest.skipUnless` (line 868) to skip the test on not-Windows.[10] Developers can also provide a reason for skipping (or not) the test. In this case, the reason specified is that the test is "*Win32 specific tests*" (line 868). The test decorator in Figure 4 is functionally equivalent to a conditional `if sys.platform == "win32"` statement but provides a more concise and declarative way to skip tests based on platform.



```
868    @unittest.skipUnless(sys.platform == "win32", "Win32 specific tests")
869    def test_stat_block_device(self):
870        # bpo-38030: os.stat fails for block devices
871        # Test a filename like "//./C:"
872        fname = "//./" + os.path.splitdrive(os.getcwd())[0]
873        result = os.stat(fname)
874        self.assertEqual(result.st_mode, stat.S_IFBLK)
```

Figure 4: Example OS-specific test in test decorator. The OS identification API `sys.platform` occurs in the test decorator.

The unittest [35] and pytest [30] documentation lists available decorators for conditionally skipping tests. There are five main decorators: two in pytest (`skipif` and `xfail`), and three in unittest (`skipIf`, `skipUnless`, and `expectedFailure`), as shown in Table 2.

Table 2: Test decorators in unittest and pytest.

Decorator	Short Description
`@unittest.skipIf`	Skip the decorated test if condition is True.
`@unittest.skipUnless`	Skip the decorated test unless condition is True.
`@unittest.expectedFailure`	Mark the test as an expected failure or error.
`@pytest.mark.skipif`	Skip a test function if a condition is True.
`@pytest.mark.xfail`	Marks a test function as expected to fail.

2.3 Related Work

Application Programming Interfaces (APIs) have been extensively studied for their impact on software reuse, developer productivity, and system maintainability. Research has addressed topics such as API migration, deprecation, misuse, and evolution [13, 14, 19, 21, 24]. Lamothe *et al.* [18] identified API maintenance and usability as core themes. However, OS identification APIs have received little attention. Additionally, test smells denote poor design practices in test code and have been shown to negatively impact maintainability and comprehension [3, 26]. Although several detection tools exist [4, 36], none specifically target OS-specific tests. Cataloged smells such as *Conditional Test Logic* and *Rotten Green Tests* are relevant, as OS-specific tests often rely on conditional structures and may result in not-executed assertions. Lastly, flaky tests exhibit non-deterministic behavior and are a major concern in regression testing [8, 20]. Platform dependency has been identified as a root cause of test flakiness, with OS-specific failures reported in both JavaScript and Android projects [11, 37]. Common mitigation strategies include test skipping or exclusion from code coverage [12, 15]. As for the OS-specific tests, Job and Hora [15] presented an empirical analysis of OS-specific tests, investigating how and why developers implement them. The study finds that such tests are common, target a diversity of code, and often involve multiple operations. The primary motivations for their implementation are to address unavailable external resources, unsupported standard libraries, and flaky tests. However, although the study provides a qualitative analysis based on a curated set of projects, the study does not address the broader importance of testing across multiple platforms, nor the specific relevance of CPython and its potential cascading effects. Furthermore, it does not explore aspects related to issues and security vulnerabilities.

3 Study Design

3.1 Selecting Software System

We selected a system implemented in Python due to its prominent position in current language popularity rankings and its mature ecosystem. We focus on CPython, the reference implementation of the Python programming language [10]. The CPython code structure provides a summary of file locations for modules and builtins [34]. Additionally, CPython is a relevant subject for empirical studies because it serves as the foundation for the entire Python ecosystem, meaning that changes, defects, or platform-specific behaviors can propagate across numerous dependent applications, libraries, and frameworks. Moreover, its cross-platform support (*e.g.,* Windows, macOS, Linux, and BSD) demands explicit design decisions and testing to address differences in APIs, behavior, and resource availability. The typical source code layout for a Python module includes the following files:

- Lib/*<module>.py*
- Modules/_*<module>.c*
- Lib/test/test_*<module>.py*
- Doc/library/*<module>.rst*

[9]https://github.com/python/cpython/blob/bbe9c31edc4fc3e1cdc908e9a06593c394f4bfdb/Lib/test/test_popen.py#L52
[10]https://github.com/python/cpython/blob/bbe9c31edc4fc3e1cdc908e9a06593c394f4bfdb/Lib/test/test_os.py#L868

We filter out 919 production files, as they are not relevant to our research. Furthermore, our study focused only on Python files. To include test files for standard modules, extension modules, built-in types, and built-in functions, we prioritized Python files in the `Lib/test/` directory. We analyzed 1,154 test files and found 156 occurrences of OS-specific tests, as described in the section 3.2. Table 3 summarizes the analyzed files and these OS-specific test occurrences in the CPython repository.

Table 3: Summary of files in CPython.

All files	2,102
Production files	919
Test files	1,183
Analyzed test files	1,154
Test files with problems	29
Test files with OS-specific tests (total)	156

3.2 Detecting OS-Specific Tests

An *OS-specific test* is a test that detects the platform on which it is executed (Section 2.2). A test file is classified as OS-specific if it calls an OS identification API (Section 2.1). For instance, the test `test_file_mode` presented in Figure 2 has a call to the API `sys.platform` which returns a string with the current platform.

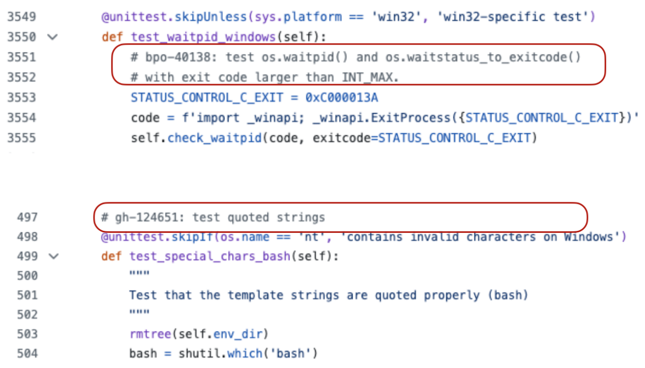
Thus, to detect OS-specific tests, we spot for occurrences of OS identification API in both test decorators and test code. To this end, we performed our automated tool for extracting OS-specific Tests from Git repositories written in Python [16]. The tool detects the usage of OS identification API in test files, both test decorators and test code. We used the OS identification API listed in Table 1. As an output, the tool exports information about OS-specific tests (*e.g.,* project name, GitHub link, OS identification API, and target platform) and stores it in two separate CSV files: one for test decorators and another for test code. Then, we ran the tool on the 1,154 test files and detected 156 OS-specific tests, containing a total of 651 OS identification API occurrences (303 in test decorators and 348 in test code). Our results are publicly available [17].

3.3 Collecting Issues

Lastly, we collected the issues related to OS-specific tests. Issues and bugs related to OS-specific tests are reported through issue trackers. Although the current issue tracker is hosted on GitHub, we also considered the legacy tracker hosted at *https://bugs.python.org*.

For this purpose, we manually inspected the source code around the OS-specific test occurrences and mapped the associated issue numbers to the corresponding issue tracker. In addition to verifying the existence of the issue number, we analyzed the issue descriptions and related discussions to confirm their relevance. Figure 5 presents two examples of this mapping. First, the issue number *bpo-40138*[11] was mapped to *https://bugs.python.org/issue40138*. In the second example, the issue number *gh-124651*[12] was mapped to *https://github.com/python/cpython/issues/124651*. Overall, we found 170 issues added together across both issue trackers.

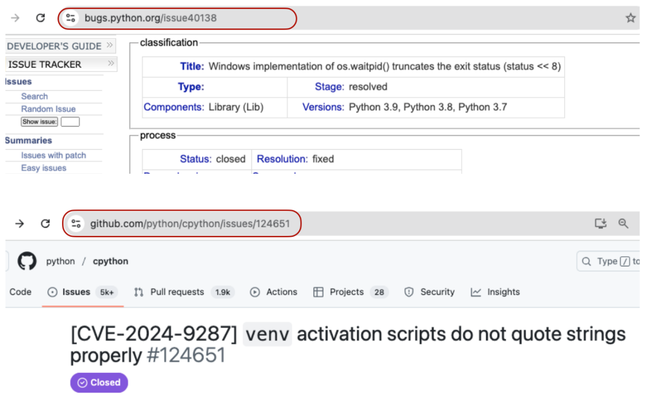
[11]https://github.com/python/cpython/blob/bbe9c31edc4fc3e1cdc908e9a06593c394f4bfdb/Lib/test/test_venv.py#L498
[12]https://github.com/python/cpython/blob/bbe9c31edc4fc3e1cdc908e9a06593c394f4bfdb/Lib/test/test_os.py#L3549



```python
    @unittest.skipUnless(sys.platform == 'win32', 'win32-specific test')
    def test_waitpid_windows(self):
        # bpo-40138: test os.waitpid() and os.waitstatus_to_exitcode()
        # with exit code larger than INT_MAX.
        STATUS_CONTROL_C_EXIT = 0xC000013A
        code = f'import _winapi; _winapi.ExitProcess({STATUS_CONTROL_C_EXIT})'
        self.check_waitpid(code, exitcode=STATUS_CONTROL_C_EXIT)
```

```python
    # gh-124651: test quoted strings
    @unittest.skipIf(os.name == 'nt', 'contains invalid characters on Windows')
    def test_special_chars_bash(self):
        """
        Test that the template strings are quoted properly (bash)
        """
        rmtree(self.env_dir)
        bash = shutil.which('bash')
```

(a) Examples of two issues number in source code (*bpo-40138* and *gh-124651*).



bugs.python.org/issue40138

classification

Title: Windows implementation of os.waitpid() truncates the exit status (status << 8)	
Type:	**Stage:** resolved
Components: Library (Lib)	**Versions:** Python 3.9, Python 3.8, Python 3.7

process

Status: closed	**Resolution:** fixed

github.com/python/cpython/issues/124651

[CVE-2024-9287] venv activation scripts do not quote strings properly #124651

(b) Example of two issues in issue tracker (*bugs.python* and *github.com*).

Figure 5: Example of two issues related to OS-specific tests. *bpo-40138* was mapped to *bugs.python.org/issue40138* and *gh-124651* mapped to *github.com/python/cpython/issues/124651*.

3.4 Research Questions

3.4.1 RQ1: How frequent are OS-specific tests in the CPython project? In this first research question, we assess the frequency of OS-specific tests in CPython. We also examine which OS identification APIs are most frequently used and which platforms are most commonly targeted. To identify the target platform, we mapped the possible value returned by the APIs, like *win32* for Windows and *darwin* for macOS. For example, Figure 5a shows the use of `sys.platform` and `os.name`, which check for `win32` and `nt`, both of which are mapped to Windows. Table 4 shows some of the possible value returned by the OS identification APIs and its respective mappings.
Rationale: We aim to better understand to what extent OS-specific tests happen in CPython project and identify the most targeted platforms. So far, the extent to which these tests are used remains unclear.

Table 4: Target platform and parameter used to mapping.

Platforms	Parameters
Windows	`win32, nt, win`
macOS	`darwin, Darwin`
POSIX	`posix`
Linux	`linux`
Android	`android`
VxWorks	`vxworks`
Cygwin	`cygwin`
Solaris	`sunos, sunos5, SunOS, solaris`
WebAssembly	`wasi, emscripten`
AIX	`aix, AIX`

3.4.2 RQ2: Where are OS identification APIs implemented in tests? In our second research question, we investigate where OS identification APIs are implemented: in (1) test code or (2) test decorators. For both locations, we collect the occurrence of OS identification API and target platforms. In particular, when it happens in test decorators, we also explore what decorators are adopted (see Table 2). **Rationale:** We aim to evaluate where OS identification API are most frequently located in tests. At this point, we are not aware of where developers use this kind of API in CPython project. This analysis may reveal important insights into the implementation of OS-specific tests.

3.4.3 RQ3: What issues are related to OS-specific tests? In last research question, we assess the frequency of issues associated with OS-specific tests in CPython. We explore what are the most issues related to OS-specific test, the OS identification APIs, and the most targeted platforms. To collect the issues, we mapped the issues number to respectively issue tracker (Section 3.3). For each occurrence of OS identification APIs, we can relate multiple issues (Figure 5). **Rationale:** Our goal is to understand the extent to which issues are related to OS-specific tests, and which platforms are most targeted. So far, it is unclear how common or why this is happening. If such tests are frequently related to issues, this may prompt further discussion about whether OS-specific tests are being created as a response to recurring platform-specific problems.

4 Results

4.1 RQ1: Frequency of OS-specific tests

Table 5 summarizes the frequency of OS-specific tests in CPython. We find OS-specific tests in 156 out of the 1,154 analyzed test files. CPython has a total of 651 occurrences of OS Identification APIs. On mean, 13.51% of test files contain OS-specific tests, with an average of 4.17 OS identification API occurrences per such test file.

We also investigate the most frequently used OS identification APIs. Table 6 presents the most used OS identification APIs. The 651 occurrences are mostly concentrated in two APIs: `sys.platform` (73.58%) and `os.name` (23.66%). Other APIs represent only 2.76% of the occurrences.

Lastly, we assess what are the most targeted platforms in the OS identification APIs. As presented in Table 7, Windows is the top one targeted platform (59.19%), followed by macOS (10.10%), and POSIX (5.05%). In contrast, Linux, Android and VxWorks happen in only 11.78% of the cases. We found 22 different platforms, among which 16 other platforms account for a total of 13.88% of cases. Table 7 shows a total of 713 of cases, which is higher than the 651 occurrences of the OS identification APIs. This discrepancy arises because one single occurrence of an OS identification API may target multiple platforms (*e.g.,* `os.name in [‘posix’, ‘nt’]`).

Table 5: Summary of OS-specific tests in CPython.

Test files Analyzed	1,154
Test files with OS-specific tests (total)	156
Test files with OS-specific tests (mean)	13.51%
Occurrences of OS identification APIs (total)	651
Occurrences of OS identification APIs (decorator)	303
Occurrences of OS identification APIs (code)	348
OS identification APIs in test files (mean)	56.41%
OS identification APIs in OS-specific tests (mean)	4.17

Table 6: Most used OS identification APIs in CPython.

Pos	API	#	%
1	`sys.platform`	479	73.58
2	`os.name`	154	23.66
3	`sys.getwindowsversion`	5	0.77
4	`os.supports_bytes_environ`	4	0.61
5	`platform.mac_ver`	3	0.46
6	`platform.system`	5	0.77
7	`platform.uname`	1	0.15
All		651	100

Table 7: Most targeted platforms.

Pos	Target platform	#	%
1	Windows	422	59.19
2	macOS	72	10.10
3	POSIX	36	5.05
4	Linux	31	4.35
5	Android	28	3.93
6	VxWorks	25	3.51
7	Others	99	13.88
All		713	100

Summary RQ1*: OS-specific tests are common in CPython, appearing in 13% of analyzed test files. Windows is the most targeted platform, while* `sys.platform` *is the most used OS identification API.*

4.2 RQ2: Location of OS identification API

In this RQ, we explore where OS-specific tests are implemented. As described in Section 2, OS-specific tests can occur in test code (for example, in `if` blocks) or test decorators (for example, in `@skipUnless` decorators). Among the 651 occurrences of OS identification API, 348 (53.46%) are located in test code and 303 in test decorators (46.54%), as presented in Table 8.

Table 8: Location of OS Identification API.

Pos	OS Identification API		Decorator		Code	
	API	#	#	%	#	%
1	`sys.platform`	479	222	46.35	257	53.65
2	`os.name`	154	78	50.65	76	49.35
3	`sys.getwindows-version`	5		0.00	5	100.00
4	`os.supports-_bytes_environ`	4		0.00	4	100.00
5	`platform.mac_ver`	3		0.00	3	100.00
6	`platform.system`	5	3	60.00	2	40.00
7	`platform.uname`	1		0.00	1	100.00
All		651	303	46.54	348	53.46

Table 9 summarizes the OS identification API by the target platform. Windows is the most targeted platform both in test code and test decorators. It alone accounts for 59.18% of all cases. In turn, POSIX and VxWorks are the two platforms that happen most in test decorators, the others happen more frequently in the test code.

Table 9: Location of OS Identification API by the target platform.

Pos	Target Platform		Decorator		Code	
	Name	#	#	%	#	%
1	Windows	422	198	46.92	224	53.08
2	macOS	72	26	36.11	46	63.89
3	POSIX	36	24	66.67	12	33.33
4	Linux	31	13	41.94	18	58.06
5	Android	28	11	39.29	17	60.71
6	VxWorks	25	16	64.00	9	36.00
7	Others	99	25	25.25	74	74.75
All		713	303		400	

We also explore the frequency of the OS identification API occurrences that happen in *test decorators*. Table 10 presents that the decorator `@unittest.skipUnless` (provided by unittest) is the most used to skip tests depending on the platform, with 51.49% of the cases. It is followed by `@unittest.skipIf`, with 48.51%. Other decorators provided by pytest like `@pytest.mark.xfail` and `@pytest.mark.skipif` are not used with the OS Identification API. Despite being a widely used framework, the exclusive use of unittest contrasts with prior studies, which suggests that pytest is currently the most used Python testing framework [2].

Table 10: Most adopted decorators.

Decorator	#	%
`@unittest.skipUnless`	156	51.49
`@unittest.skipIf`	147	48.51
All	303	100.00

> ***Summary RQ2****: OS Identification APIs in tests are used more frequently in test code (53.46%) than in test decorators (46.54%). The decorator* `@unittest.skipUnless` *is the most used to skip tests depending on the platform. Decorators provided by pytest are not used with the OS Identification API.*

4.3 RQ3: Issues and OS-specific tests

Finally, we analyze the frequency of the issues related to OS-specific tests. Among the 170 issues related to OS-specific tests, we found that 91 issues are related in decorator, while 79 happen in the test code. Table 11 shows the most issues related to OS-specific tests, the target platform, and the location of the OS identification API occurrences. The issue `bpo-43884`[13] is highest frequent (4), followed by `bpo-29972`[14] (4) and `gh-124651`[15] (3). The prefix (*bpo* or *gh*) indicates that issue occurs in *bugs.python* or *Github*, respectively.

Table 11: Examples of issues related to OS-specific tests.

Pos	Issues	Target Platform	Decorator	Code	Total
1	bpo-43884	Windows	0	4	4
2	bpo-29972	aix	4	0	4
3	gh-124651	Windows	3	0	3
4	bpo-36819	Windows	0	3	3
5	gh-82300	Posix	2	0	2
6	Others		82	72	154
All			91	79	170

We also explore what are the most targeted platforms in the issues occurrences. Table 12 presents the top-5 most target platforms. Again, Windows (47.85%), macOS (10.75%), and POSIX (5.91%) are the most target platforms, together representing 64.52%.

Finally, we investigate where the issues are located: in test decorator or in test code. Table 13 details the frequency of issues related to OS-specific tests by OS identification API location. Although OS identification APIs occur more frequently in test code (348 occurrences), issues are more common in test decorators (27.06%). Overall, issues happen in 22.89% of OS identification API occurrences, being more concentrated in test decorators with 27.06%, while in test code they occur in 19.25% of cases. Note that each OS occurrence of the OS identification API may be related to more than one issue, and because of this we have this difference between distinct issues (170, see Table 11) and issue by OS identification API location (149, see Table 13).

[13]https://bugs.python.org/issue43884
[14]https://bugs.python.org/issue29972
[15]https://github.com/python/cpython/issues/124651

Table 12: Frequency of issues related to OS-specific tests by the target platform.

Pos	Target Platform	#	%
1	Windows	89	47.85
2	macOS	20	10.75
3	POSIX	11	5.91
4	VxWorks	11	5.91
5	AIX	10	5.38
6	Others	45	24.19
All		186	100.00

Table 13: Frequency of issues related to OS-specific tests by OS Identification API location.

Pos	Location	Occurrences	Issues	
		Total	#	%
1	Decorator	303	82	27.06
2	Code	348	67	19.25
All		651	149	22.89

> ***Summary RQ3****: Issues are related to OS-specific tests in the CPython. We find 170 issues related to OS-specific tests in CPython. 22.89% of the OS identification APIs have associated issues, and Windows is the most targeted platform (47%).*

5 Discussion and Implications

Importance of testing on multiple platforms. Overall, we find that OS-specific tests are common in the CPython project. For instance, RQ1 presented that 13.52% of the analyzed test files are OS-specific tests. The literature reports that this phenomenon is also common in other Python projects [14, 15]. Although Windows is the most frequently targeted platform, we found 22 different platforms, which reinforces the need for practitioners to maintain OS-specific tests. However, it remains unclear what strategies are used and how practitioners can best report the OS-specific tests. Likewise, based on our findings, researchers can direct their research and investigate why Windows is the most commonly targeted platform for OS-specific tests, or other mechanisms to identify the platform.
Importance of CPython and possible cascading effects. The CPython is the reference implementation of the Python programming language and is widely used across of systems. Consequently, errors in CPython may propagate across many dependent systems and platforms. As observed in RQ1, tests target 22 different platforms, which suggests a broad ecosystem impact. Table 14 presents these platforms found. Prior studies on Java ecosystems (*e.g.,* Maven Central) have explored similar propagation effects [5, 32, 33]. These studies emphasize that modern software development increasingly relies on reusable libraries and components, managing dependencies has become critical for ensuring software stability and security [5]. However, challenges like complexity of interdependent libraries can significantly impact project maintenance [5], and that even smaller, less connected libraries can trigger significant cascading effects through complex dependency chains, demonstrating ecosystem vulnerability extends beyond just critical components [33]. Moreover, artifacts with more frequent releases tend to exhibit fewer known vulnerabilities [31]. Our findings suggest that OS-specific testing practices in CPython could also play a role in such propagation patterns, given the importance of this project. Future research directions include studies that can explore how CPython-dependent software may be critically affected by effects and how OS-specific testing may be associated with these effects.

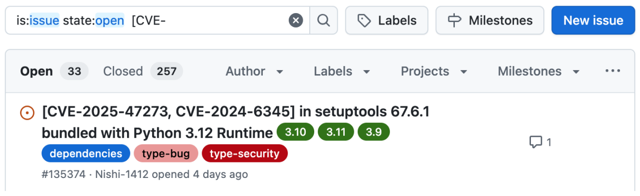
Table 14: Target platforms found in CPython test files.

Linux, Windows, macOS, POSIX, AIX, Android
WebAssembly, Solaris, VxWorks, Cygwin , OpenBSD
FreeBSD, GNU/kFreeBSD , DragonFlyBSD, NetBSD
HP-UX, OpenVMS, OSF/1, UnixWare, ios, tvos, watchos

Issues associated with OS-specific tests. In RQ3, we detected that issues are related to OS-specific tests in the CPython project. In particular, we found 170 issues, of which 91 are related to the test decorator, while 79 occur in the test code. Moreover, 27.06% of the test decorators and 19.25% of the test code are directly associated with at least one issue. It is important to notice that although we have detected a considerable number of issues, only three are open: `gh-110012`,[16] `gh-77631`,[17] and `bpo-46390`.[18] However, the direction of this relationship remains unclear. It is not evident whether issues trigger the creation of OS-specific tests or whether these tests themselves introduce or reveal issues. Whether the issues motivate the creation of OS-specific tests, or vice versa. We encourage future work to explore this causal relationship across other projects.
Issues and security vulnerabilities. Modern software development landscape heavily relies on transitive dependencies, which simplify integration but introduce hidden risks [23, 29, 31]. The CVE program is designed to assign unique identifiers to known vulnerabilities and associate them with specific software versions [6]. A single vulnerable library in the dependency graph can compromise an entire project through CVEs. Prior work shows that increasing dependency depth prolongs vulnerability resolution time, while frequent updates reduce exposure [29, 31]. As discussed in RQ3, several issues in CPython tests are related to OS-specific tests. In particular, one case deserves to be highlighted, as the issue (*gh-124651*[19]) is associated with a Common Vulnerabilities and Exposures (CVE). Such vulnerabilities, when exploited, can compromise confidentiality, integrity, or availability [25]. We identified this relationship, inspecting the issue title and look for term *CVE*. In the issue *gh-124651*, the title is *[CVE-2024-9287] venv activation scripts do not quote strings properly.* [20] Thus, we manually look at how

[16]https://github.com/python/cpython/issues/110012
[17]https://github.com/python/cpython/issues/77631
[18]https://bugs.python.org/issue46390
[19]https://github.com/python/cpython/issues/124651
[20]The `CVE-2024-9287` is described like a vulnerability has been found in the CPython 'venv' module and CLI where path names provided when creating a virtual environment were not quoted properly, allowing the creator to inject commands into virtual environment "activation" scripts. Availability at https://nvd.nist.gov/vuln/detail/cve-2024-9287

many issues are related to the term "CVE"[21] and identified 290 CVE-related issues in CPython (33 open and 257 closed), as shown in Figure 6. This association underscores the relevance of OS-specific tests in security-sensitive contexts and the potential impact they may have on the broader software ecosystem. Given that we identified issues related to OS-specific tests (RQ3), we emphasize the need for security-conscious release processes. This includes prioritizing rapid patching of CVE-related issues and developing automated tools to detect OS-specific vulnerabilities.


is:issue state:open [CVE-

Open 33 Closed 257

[CVE-2025-47273, CVE-2024-6345] in setuptools 67.6.1 bundled with Python 3.12 Runtime 3.10 3.11 3.9

dependencies type-bug type-security

#135374 · Nishi-1412 opened 4 days ago

Figure 6: Distribution of CVE-related issues in CPython: 33 open and 257 closed.

6 Threats to Validity

Selection of issues related to OS-specific tests. To detect API to identify issues, we have manually inspected the source code of OS-specific test. We ended up selecting 170 issues added together across both issue trackers. As this selection is purely based on reading the source code, identification of issue number around the OS identification API occurrences, and looking the issue number in issue tracker, the risks of false positives (wrong issues) and false negatives (missing issues) are reduced. To minimize incorrect mappings, after identifying the issue number, we analyzed whether the issue is directly related to the occurrence of the OS identification API.

Generalization of the results. This study focuses exclusively on CPython, the reference implementation of the Python programming language. While CPython is widely used and influential, the results presented here may not generalize to other Python projects or to systems developed in different programming languages. Future research should replicate this analysis in other software ecosystems to better understand the prevalence and nature of OS-specific testing practices in diverse development environments.

Number analyzed tests files. In the our research questions, we quantitatively analyzed all instances of OS-specific tests detected in the project, in 1,154 test files. That is, it was not necessary to select a sample, since all instances were analyzed. For a large number of tests, we recommend that researchers randomly select statistically significant samples.

7 Conclusion and Further Steps

In this paper, we presented an empirical study investigating how developers implement OS-specific tests in CPython project. We examined this project and assessed their OS-specific tests quantitatively in three research questions. Our main results can be summarized as follows:

- **RQ1:** OS-specific tests are common in the CPython project. We find that 13% of the analyzed test files are OS-specific tests. `sys.platform` is the most used OS identification API, while Windows is the most targeted platform;
- **RQ2:** OS Identification APIs in tests are used more frequently in test code (53.46%) than in test decorators (46.54%). The decorator `@unittest.skipUnless` is the most used to skip tests depending on the platform; and
- **RQ3:** We find 170 issues related to OS-specific tests in CPython. Test decorators are the most related to issues (27.06%).

Lastly, we discussed practical implications for practitioners and researchers. Based on our findings, we discussed the importance of testing on multiple platforms, the relationship between issues and related to OS-specific tests, the cascading effects that may occur with CPython-dependent systems, and we relate the issues to vulnerabilities. As future work, we plan to extend this study by analyzing the full set of issues in the project, including historical data from version control. This will allow us to better understand how the relationship between OS-specific tests and issues evolves over time. This knowledge would provide the basis for better understanding how developers implement OS-specific tests, possibly supporting the development of a catalog of the motivations, challenges, and solutions adopted.

ACKNOWLEDGMENTS

This research is supported by CAPES, CNPq, and FAPEMIG.

REFERENCES

[1] Vincent Aranega, Julien Delplanque, Matias Martinez, Andrew P. Black, Stéphane Ducasse, Anne Etien, Christopher Fuhrman, and Guillermo Polito. 2021. Rotten green tests in Java, Pharo and Python. *Empirical Software Engineering* 26, 6 (2021), 130.
[2] Lívia Barbosa and Andre Hora. 2022. How and why developers migrate Python tests. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 538–548.
[3] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. 2012. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *International Conference on Software Maintenance (ICSM)*. 56–65.
[4] Alexandru Bodea. 2022. Pytest-Smell: A Smell Detection Tool for Python Unit Tests. In *International Symposium on Software Testing and Analysis*. ACM, 793–796.
[5] Barisha Chowdhury, Md Fazle Rabbi, S. M. Mahedy Hasan, and Minhaz F. Zibran. 2025. Insights into Dependency Maintenance Trends in the Maven Ecosystem . In *International Conference on Mining Software Repositories (MSR)*. 280–284.
[6] CVE - Common Vulnerabilities and Exposures. June, 2025. https://www.cve.org/.
[7] Julien Delplanque, Stéphane Ducasse, Guillermo Polito, Andrew P. Black, and Anne Etien. 2019. Rotten Green Tests. In *International Conference on Software Engineering (ICSE)*. IEEE, 500–511.
[8] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding Flaky Tests: The Developer's Perspective. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 830–840.
[9] GitHub-hosted runners. June, 2025. https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners/about-github-hosted-runners.
[10] CPython Glossary. June, 2025. https://docs.python.org/3/glossary.html-CPython.
[11] Negar Hashemi, Amjed Tahir, and Shawn Rasheed. 2022. An Empirical Study of Flaky Tests in JavaScript. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 24–34.
[12] Andre Hora. 2023. Excluding code from test coverage: practices, motivations, and impact. *Empirical Software Engineering* 28, 1 (2023), 1–33.
[13] Andre Hora, Romain Robbes, Marco Tulio Valente, Nicolas Anquetil, Anne Etien, and Stephane Ducasse. 2018. How do Developers React to API Evolution? A Large-Scale Empirical Study. *Software Quality Journal* 26, 1 (2018), 161–191.

[21] https://github.com/python/cpython/issues?q=is%3Aissue%20state%3Aopen%20%20%5BCVE-

[14] Ricardo Job and Andre Hora. 2024. Availability and Usage of Platform-Specific APIs: A First Empirical Study. In *International Conference on Mining Software Repositories*. 27–31.

[15] Ricardo Job and Andre Hora. 2024. How and Why Developers Implement OS-Specific Tests. *Empirical Software Engineering* 30 (2024), 33.

[16] Ricardo Job and Andre Hora. 2025. *OSTDetector: An automated tool for extracting OS-specific Tests from Git repositories written in Python.* https://doi.org/10.5281/zenodo.10120045

[17] Ricardo Job and Andre Hora. July, 2025. *On the Implementation of OS-Specific Tests: The CPython Case.* https://doi.org/10.5281/zenodo.15794483

[18] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. 2021. A Systematic Review of API Evolution Literature. *ACM Computing Surveys (CSUR)* 54, 8 (2021), 1–36.

[19] Can Li, Jingxuan Zhang, Yixuan Tang, Zhuhang Li, and Tianyue Sun. 2024. Boosting API Misuse Detection via Integrating API Constraints from Multiple Sources. In *International Conference on Mining Software Repositories*. 14–26.

[20] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *International Symposium on Foundations of Software Engineering*. ACM, 643–653.

[21] Matias Martinez and Bruno Gois Mateus. 2022. Why Did Developers Migrate Android Applications From Java to Kotlin? *IEEE Transactions on Software Engineering* 48 (2022), 4521–4534.

[22] Gerard Meszaros. 2007. *xUnit test patterns: Refactoring test code.* Pearson Education.

[23] Costain Nachuma, Md Mosharaf Hossan, Asif K. Turzo, and Minhaz F. Zibran. 2025. Decoding Dependency Risks: A Quantitative Study of Vulnerabilities in the Maven Ecosystem . In *International Conference on Mining Software Repositories (MSR)*. 270–274.

[24] Romulo Nascimento, Eduardo Figueiredo, and Andre Hora. 2021. JavaScript API Deprecation Landscape: A Survey and Mining Study. *IEEE Software* 39, 3 (2021), 96–105.

[25] National Institute of Standards and Technology. June, 2025. https://nvd.nist.gov/vuln.

[26] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. 2016. On the Diffusion of Test Smells in Automatically Generated Test Code: An Empirical Study. In *International Workshop on Search-Based Software Testing*. ACM, 5–14.

[27] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2019. On the distribution of test smells in open source Android applications: an exploratory study. In *International Conference on Computer Science and Software Engineering*. IBM Corp., 193–202.

[28] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2020. TsDetect: An Open Source Test Smells Detection Tool. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 1650–1654.

[29] Piotr Przymus, Mikolaj Fejzer, Jakub Narebski, Krzysztof Rykaczewski, and Krzysztof Stencel. 2025. Out of Sight, Still at Risk: The Lifecycle of Transitive Vulnerabilities in Maven . In *International Conference on Mining Software Repositories (MSR)*. 329–333.

[30] Pytest. June, 2025. https://docs.pytest.org.

[31] Md Shafiullah Shafin, Md Fazle Rabbi, S. M. Mahedy Hasan, and Minhaz F. Zibran. 2025. Faster Releases, Fewer Risks: A Study on Maven Artifact Vulnerabilities and Lifecycle Management . In *International Conference on Mining Software Repositories (MSR)*. 275–279.

[32] Mehedi Hasan Shanto, Muhammad Asaduzzaman, Manishankar Mondal, and Shaiful Chowdhury. 2025. Dependency Dilemmas: A Comparative Study of Independent and Dependent Artifacts in Maven Central Ecosystem . In *International Conference on Mining Software Repositories (MSR)*. 304–308.

[33] Mina Shehata, Saidmakhmud Makhkamjonoov, Mahad Syed, and Esteban Parra. 2025. Cascading Effects: Analyzing Project Failure Impact in the Maven Central Ecosystem . In *International Conference on Mining Software Repositories (MSR)*. 309–313.

[34] CPython source. June, 2025. https://devguide.python.org/internals/exploring.

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

[36] Tongjie Wang, Yaroslav Golubev, Oleg Smirnov, Jiawei Li, Timofey Bryksin, and Iftekhar Ahmed. 2021. PyNose: A Test Smell Detector For Python. In *International Conference on Automated Software Engineering (ASE)*. IEEE, 593–605.

[37] Hao Xia, Yuan Zhang, Yingtian Zhou, Xiaoting Chen, Yang Wang, Xiangyu Zhang, Shuaishuai Cui, Geng Hong, Xiaohan Zhang, Min Yang, et al. 2020. How Android developers handle evolution-induced API compatibility issues: a large-scale study. In *International Conference on Software Engineering*. 886–898.