PSASpotter: A Tool to Detect the Usage of Platform-Specific APIs in Python

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

Andre Hora
DCC, UFMG
Belo Horizonte, Brazil
andrehora@dcc.ufmg.br

***Abstract*—A *platform-specific API* is implemented for a particular platform (*e.g.,* operating system), thus, it may not work on other platforms than the target one. Detecting the usage of such APIs is important for supporting software maintenance, as it allows maintainers to be alerted about APIs that could pose potential risks. This paper proposes PSASpotter, a tool to detect the usage of platform-specific APIs in Python systems. PSASpotter also identifies whether the platform-specific APIs are used within a defensive code, such as `try/except` blocks or `if` blocks that check the current platform. PSASpotter can support the development of novel empirical studies about the usage of platform-specific APIs in the Python ecosystem. Moreover, the defensive code detected by PSASpotter may contain alternative solutions for unavailable APIs, which can provide insights for software development and testing across multiple platforms. PSASpotter is available at: https://github.com/ricardojob/PSASpotter. Tool video: https://youtu.be/d3WyozTAKS8.**

***Index Terms*—Platform-Specific APIs, Defensive Code, Python, Software Maintenance**

I. Introduction

A *platform-specific API* is an API implemented for a particular platform (*e.g.,* operating system) and therefore may not work on other platforms than the target one [10]. Such limitations are typically associated with *availability restrictions*, which, in this work, refer to conditions that limit the execution or availability of an API according to the underlying platform. In Python, many APIs provided by the Python Standard Library have availability restrictions [10]. For example, the API `os.pathconf`[1] is available for Unix, while the API `_thread.stack_size`[2] is available for Windows.

A system that calls platform-specific APIs may implement defensive code to ensure it works properly on the desired platforms, such as `try/except` blocks or `if` blocks that check the API availability. For instance, Figure 1a presents an example where the Unix API `os.pathconf` is called within an `if` block that checks whether the current OS has such API.[3] Figure 1b shows another case in which the Windows API `_thread.stack_size` is called, but the current platform is not checked.[4] As a result, this code might not work correctly when executed on Unix-like platforms.

```
169  def _get_max_path_length() -> int:
170      if hasattr(os, "pathconf"):
171          return os.pathconf("/", "PC_PATH_MAX")
172      # Windows
173      return _DEFAULT_WIN_MAX_PATH_LENGTH
```

(a) Unix-specific API `os.pathconf`.

```
72   def test_stack_size(self):
73       # Various stack size tests.
74       self.assertEqual(thread.stack_size(), 0, "initial stack size is not 0")
75
76       thread.stack_size(0)
77       self.assertEqual(thread.stack_size(), 0, "stack_size not reset to default")
```

(b) Windows-specific API `_thread.stack_size`.

Fig. 1: Examples of the platform-specific APIs in Python.

Detecting the usage of platform-specific APIs is important for supporting software maintenance, as it allows project maintainers to be alerted about APIs that could pose potential risks. However, this is not a trivial task because it requires (i) a comprehensive understanding of all platform-specific APIs and (ii) a tool to automatically identify them in source code. In a prior study [10], we found that the Python Standard Library [25] has over 1,800 platform-specific APIs spread across 17 platforms, such as Linux, Windows, macOS, Unix, and Android. We also found that platform-specific APIs are widely used in the Python ecosystem, with over 19K usages of 683 platform-specific APIs in the 100 analyzed projects [10].

To overcome this gap, this paper proposes PSASpotter,[5] a tool to automatically detect the usage of platform-specific APIs in Python systems (Section II). PSASpotter also identifies whether the platform-specific APIs are used within defensive code, such as `try/except` blocks or `if` blocks that check the current platform. Next, we discuss practical applications of tool (Section III). Finally, we evaluate the tool's performance in detecting whether platform-specific API usage occurs within defensive code (Section IV). The tool achieves precision, recall, and accuracy of over 97%, 83%, and 87%, respectively.

PSASpotter can support the development of novel empirical studies about the usage of platform-specific APIs. To support

[1]https://docs.python.org/3/library/os.html#os.pathconf

[2]https://docs.python.org/3/library/_thread.html#thread.stack_size

[3]https://github.com/ray-project/ray/blob/fc98a5f286877ce7f6241961aca0c9127bee21ad/python/ray/tune/experiment/trial.py#L169

[4]https://github.com/oracle/graalpython/blob/72c38809e2a4b1b8aeab475217fd4fb6b39ccea3/graalpython/lib-python/3/test/test_thread.py#L72

[5]PSASpotter is available at: https://github.com/ricardojob/PSASpotter.

such studies, we mined the usage of platform-specific APIs in 9,205 Python repositories and created a large-scale dataset. Additionally, the defensive code detected by PSASpotter may contain alternative solutions for unavailable APIs. These alternative solutions can provide insights for software development and testing across multiple platforms.

Novelty: To our knowledge, this is the first tool to detect the usage of platform-specific APIs. We also provide three practical applications and evaluate its performance in identifying platform-specific API usage within defensive code.

II. PSASpotter

A. Overview

PSASpotter is a tool to automatically detect the usage of platform-specific APIs in Python systems. Specifically, it identifies 1,841 platform-specific APIs from the Python Standard Library [25]. These APIs were derived from our previous research, where we mined the entire Python documentation to identify platform-specific APIs [10]. The tool can be used via the command line, receiving the system to be analyzed as input and providing the API usage occurrences as output.

B. Detecting Platform-Specific API Usage

PSASpotter is an AST-based tool that detects API usage at the function/method level. Given a Git project, PSASpotter analyzes all Python files, converts them into an AST (Abstract Syntax Tree), identifies calls to platform-specific APIs, and exports details of their usage. Specifically, for each usage, PSASpotter reports information about the analyzed project (name and commit), the used API (name and availability), and the usage location (filename, line, and GitHub link[6]). In addition, PSASpotter reports whether the usage occurs within defensive code. Figure 2 provides an overview of the tool for detecting platform-specific API usages.

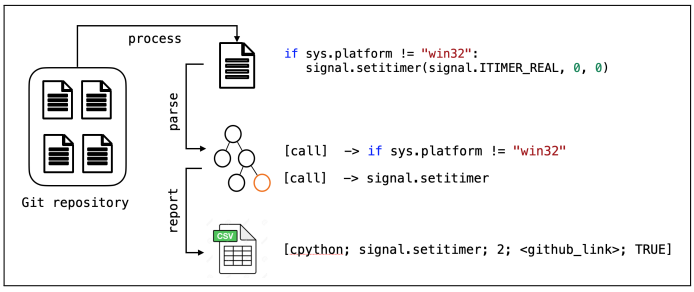

Fig. 2: Overview workflow of the PSASpotter.

C. Detecting Usage within Defensive Code

PSASpotter also identifies whether platform-specific APIs appear within defensive code, checking four cases:

- **`try/except` blocks:** Platform-specific APIs are called within `try/except` blocks. This happens when developers want to avoid any exceptions caused by calling the API on the wrong platform (*e.g.,* Figure 3a).
- **`if` blocks that check current platform:** Platform-specific APIs are called within `if` blocks that identify the current platform (such as Unix or Windows). In this case, we consider all 15 APIs available in the Python Standard Library to identify the current platform [11], such as `sys.platform`[7] and `os.name`[8] (*e.g.,* Figure 3b).
- **`if` blocks that verify API existence:** Platform-specific APIs are called within `if` blocks that check the existence of the API in the current platform. In this case, we consider the built-in function `hasattr`[9] (*e.g.,* Figure 3c).
- **`@skipif` decorators in tests:** When the usage of platform-specific APIs is located in tests, PSASpotter verifies whether these tests are annotated with decorators to conditionally skip (*e.g.,* `@skipif`) their execution under specific conditions. Using such decorators also represents a form of defensive programming, as they allow tests to be conditionally skipped on platforms where they are not supported. The tool supports five decorators, two in pytest and three in unittest: `@pytest.mark.skipif`, `@pytest.mark.xfail`, `@unittest.skipIf`, `@unittest.skipUnless`, and `@unittest.expectedFailure`.

```
76  def _get_user_id():
77      """Get the ID of the user for the current run."""
78      try:
79          return pwd.getpwuid(os.getuid())[0]
80      except ImportError:
81          return _DEFAULT_USER_ID
```

(a) Platform-specific API `os.getuid` used within a `try/except` block.

```
24      def sigterm_handler(*args):
25          # Give a one-second grace period for other processes to clean up
26          time.sleep(SIGTERM_GRACE_PERIOD_SECONDS)
27          # SIGKILL the pgroup (including ourselves) as a last-resort.
28          if sys.platform == "win32":
29              atexit.unregister(sigterm_handler)
30              os.kill(0, signal.CTRL_BREAK_EVENT)
31          else:
32              os.killpg(0, signal.SIGKILL)
```

(b) Platform-specific API `os.kill` used within an `if` block that checks the current platform. In this case, `sys.platform` is used to verify if the OS is Windows.

```
137  def get_filename_max_length(directory: str) -> int:
138      max_len = 255
139      if hasattr(os, 'pathconf') and 'PC_NAME_MAX' in os.pathconf_names:
140          max_len = os.pathconf(directory, 'PC_NAME_MAX')
141      return max_len
```

(c) Platform-specific API `os.pathconf` used within an `if` block that checks the existence of the API in the current platform. In this case, `hasattr` is used to verify if `os` has the attribute `pathconf`.

Fig. 3: Platform-specific APIs usage within defensive code.

[6]Example of GitHub link: https://github.com/ray-project/ray/blob/fc98a5f286877ce7f6/python/ray/_private/ray_process_reaper.py#L30

[7]https://docs.python.org/3/library/sys.html#sys.platform
[8]https://docs.python.org/3/library/os.html#os.name
[9]https://docs.python.org/3/library/functions.html#hasattr

D. Tool Usage

PSASpotter can be used via the command line. First, we install it via pip (`pip install psaspotter`). Next, we can use the tool to extract information about the usage of platform-specific APIs in a Git repository:

```
# Running PSASpotter
$ psaspotter <git_repo> -o <output_file.csv>
```

The first parameter is the Git repository to be analyzed. The parameter `-o` defines the name of the CSV output file, while the parameter `-c` specifies a commit or tag. For example, we can analyze a specific commit or tag with the parameter `-c`. PSASpotter also supports filtering APIs by category of platform-specific APIs using the parameter `-p`, such as main OSs (Linux, Windows, macOS, and Unix) and proprietary OSs (Solaris, AIX, and VxWorks). To improve maintainability, the tool uses a predefined list of platform-specific APIs from previous research by default, but it also supports the parameter `-f`, which allows users to provide a JSON file with the list of platform-specific APIs to be detected.

III. Practical Applications

A. Novel Empirical Studies and Datasets

We foresee that PSASpotter can support novel empirical studies about the usage of platform-specific APIs. For instance, it can reveal the diversity of platform-specific API usage and its potential risk when not used with proper defensive code.

In our previous research [10], we proposed an empirical study on the availability and usage of platform-specific APIs in 100 Python projects. Leveraging PSASpotter, we substantially expanded this dataset to 9,205 Python repositories. Table I shows some statistics about the new dataset, which contains 709,191 usages of platform-specific APIs, including 465,848 (66%) in production files and 243,343 (34%) in test files.

TABLE I: Dataset statistics.

Data	Value
GitHub repositories	9,205
Usage Platform-Specific APIs: all files	709,191
Usage Platform-Specific APIs: production files	465,848
Usage Platform-Specific APIs: test files	243,343
Processed Files: all files	1,953,333
Processed Files: production files	1,449,103
Processed Files: test files	504,230

Application 1: PSASpotter can support the development of novel empirical studies about the usage of platform-specific APIs in Python. To support such studies, we mined the usage of platform-specific APIs in 9,205 Python repositories and created a large-scale dataset, which is publicly available at: https://doi.org/10.5281/zenodo.17857593.

B. Detect Alternative Platform-Specific APIs

When calling platform-specific APIs within defensive code, developers may provide an alternative solution in case the API is unavailable. For example, they may call a third-party API or implement in-house solutions to handle the missing API. These alternative solutions can serve as a valuable resource for identifying and addressing potential challenges when developing and testing across multiple platforms. For instance, in project FaceSwap, the method `is_admin`[10] calls the Unix-specific API `os.getuid`[11] in a `try` block, as presented in Figure 4. In case an exception is raised, the alternative Windows API `ctypes.windll.shell32.IsUserAnAdmin` is used instead.[12] Finally, Figure 3b presents another potential alternatives: `os.kill` and `os.killpg`.

```
118         def is_admin(self) -> bool:
119             """ Check whether user is admin """
120             try:
121                 retval = os.getuid() == 0  # type: ignore
122             except AttributeError:
123                 retval = ctypes.windll.shell32.IsUserAnAdmin() != 0
124             return retval
```

Fig. 4: Alternative to `os.getuid` (FaceSwap).

Application 2: The defensive code detected by PSASpotter may contain alternative solutions for unavailable APIs. These alternative solutions can provide insights for identifying and resolving potential challenges encountered during development and testing across multiple platforms.

C. Identify Libraries and Frameworks with Dependence on Platform-Specific APIs

Detecting the usage of platform-specific APIs may be particularly important for libraries and frameworks. These projects are typically executed and tested on multiple platforms by their users. Therefore, having defensive code when using platform-specific APIs is fundamental to reducing the chances that errors propagate to users. As an example, Table II presents five highly relevant[13] libraries and frameworks in the Python ecosystem with their number of dependent repositories in GitHub and usage of platform-specific APIs.

Application 3: PSASpotter can identify the usage of platform-specific APIs in libraries and frameworks, helping developers decide whether such calls should rely on defensive programming and potentially reducing the chances that errors propagate to users.

[10]https://github.com/deepfakes/faceswap/blob/cbaad146d5aca9bd714bb6c69b10dd7c02d88f9d/setup.py#L121

[11]https://docs.python.org/3/library/os.html#os.getuid

[12]https://docs.python.org/3.11/library/ctypes.html

[13]https://lp.jetbrains.com/python-developers-survey-2023/

TABLE II: Usage of platform-specific APIs in selected Python projects.

Project	Dependent Repositories	Usage of Platform-Specific APIs
django/django	1,732,014	305
psf/requests	3,086,331	96
fastapi/fastapi	436,052	24
pallets/flask	2,075,961	10
encode/django-rest-framework	710,636	4

TABLE III: Performance of defensive code identification.

Project	Correct	Incorrect	Prec.	Rec.	Acc.
All	1,350	194	97.49	83.96	87.44
Test	491	76	97.07	84.88	86.60
Production	859	118	97.79	83.33	87.92
saltstack/salt	214	75	99.21	63.00	74.05
ansible/ansible	166	7	97.65	94.32	95.95
ray-project/ray	107	11	91.53	90.00	90.68
kovidgoyal/kitty	64	2	98.04	98.04	96.97
spotify/luigi	51	6	92.00	95.83	89.47

Precision (Prec.), Recall (Rec.), and Accuracy (Acc.)

IV. Tool Performance

PSASpotter identifies whether these APIs are used outside defensive code structures. To evaluate this capability, we assess the performance of the tool in identifying whether platform-specific API usage occurs within defensive code.

A. Case Study

We selected 100 popular Python real-world software systems projects hosted on GitHub. We relied on our prior dataset to collect software projects [10], [13]. We chose this dataset of projects due to the following characteristics: (1) they are written in Python, (2) they are real-world software systems, and (3) they use platform-specific APIs. This dataset includes widely used projects, such as Numpy, Django, and Pandas.

B. Manual Flagging

Initially, we collected 1,544 occurrences used by the selected client systems. Among these occurrences, 977 were located in the production code, while 567 happened in the test code. Next, for each file where an occurrence appeared, we manually examined and classified based on the usage context of platform-specific API. This process resulted in 626 instances identified as occurring in defensive code, while 918 occcurred without defensive code. We analyzed whether each file contained protective handling around the function where the API call occurs. This evaluation is publicly available at: https://doi.org/10.5281/zenodo.14018574

C. Evaluation: Precision, Recall, and Accuracy

Table III details the precision, recall, and accuracy of the 1,544 occurrences. It also shows metrics for the top-5 projects with the most uses. We manually flagged 626 occurrences as defensive code and 918 as the absence of defensive code. After running the tool, we observed a large number of false negatives (171) and few false positives (23). Regarding the correct predictions, we observed 895 true positives and 455 true negatives. Overall, we find a precision of 97.49%, a recall of 83.96%, and an accuracy of 87.44% for defensive code identification. The overall precision close to 98% indicates that PSASpotter makes correct predictions in nearly all cases. Note that the overall recall is not low (83.96%), indicating that it often correctly identify defensive code. Finally, the accuracy at 87.44% shows that the tool provides good, reliable predictions.

Results: PSASpotter has a precision of 97.49%, recall of 83.96%, and accuracy of 87.44% to identifying whether the platform-specific API usage occurs within defensive code.

D. Main Reasons of Divergence

Our findings show that PSASpotter performs well in detecting whether platform-specific API usage occurs within defensive code. To understand the discrepancies between manual inspection and the tool's output, we examined the 194 conflicting cases and identified three main reasons for divergence (another 26 cases are related to manual flagging).

In-house solutions. In some cases, projects employ in-house solutions to identify the operating system. We identified 34 involving internal decorator and 47 occurrences involving internal APIs, implemented specifically for each project (*i.e.,* `@t.skip.if_win32`[14] and `salt.utils.platform.is_windows`[15]).

Distinct values due to platform-specific APIs. We also found 21 occurrences involving value assignments, 17 involving value access, and 18 related to handling within the caller function. This reason is further supported by existing literature [11]. These cases involve control flow data that would require dynamic analysis for accurate detection.

Guard clauses. We also found 31 occurrences related to the guard clause. The tool was unable to link the API calls to the guard clause because the calls did not occur within the same `if` block. Notice that these cases are very rare.

V. Limitations

PSASpotter works for Python systems, and the platform-specific APIs come from the Python Standard Library [25]. Such library, which describes the standard library distributed with Python, is very extensive and offers a wide range of facilities [25]. Thus, it is used by virtually every Python application. Despite that, other sources of platform-specific APIs may exist in the Python ecosystem. Further studies could investigate other ways to identify the current platform, for example, from popular libraries and frameworks. Therefore,

[14]https://github.com/celery/celery/blob/fe762c3a26e56ff34608244fc04336b438f8fa0c/t/unit/apps/test_multi.py#L389

[15]https://github.com/saltstack/salt/blob/53db41263216ba00a94e10fa2fceefa732f2b776/salt/master.py#L1241

PSASpotter can be extended by incorporating more platform-specific APIs from additional sources and improving accuracy in identifying usage within defensive code. Moreover, further tools can be built to explore platform-specific APIs in other programming languages and analyze dynamic API calls.

VI. Related Work

Application Programming Interface (API) is a widely studied research topic in software engineering [14], [19], [20], [22], [23]. Prior research has empirically examined API evolution, breaking changes, deprecation, and migration [24], [2], [7], [15], [16], [17], [21], [26], [27], [1], [18]. Several tools have also been proposed to support API maintenance and evolution, including SemDiff [4], apiwave [8], APIDiff [3], Aexpy [5], REPFINDER [9], and Deprewriter [6], to name a few. However, none of these studies or tools address the detection or analysis of platform-specific API usage, leaving the frequency, scope, and risks associated with this phenomenon unexplored. To address this gap, PSASpotter can support investigations into the extent to which platform-specific APIs are handled and used.

In the context of platform-specific APIs, we have empirically explored their availability and usage in client systems [10]. Recently, we investigated OS-specific tests, namely, a test that identifies the operating system on which they are executed [11], [12]. However, neither line of research provides automated support for detecting code dealing with platforms.PSASpotter contributes to the API literature by providing an automatic tool for detecting the usage of platform-specific APIs and supporting novel research on this topic.

VII. Conclusion

This paper proposed PSASpotter, a tool to detect the usage of platform-specific APIs in Python systems. PSASpotter also identifies whether the platform-specific APIs are used within defensive code. In addition, we provided three practical applications for the usage of this tool. Moreover, we evaluated the performance of the tool in identifying whether the platform-specific API usage occurs within defensive code. Lastly, we discussed practical applications related to development and testing across multiple platforms.

Acknowledgments

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

References

[1] Lívia Barbosa and Andre Hora. How and why developers migrate python tests. In *International Conference on Software Analysis, Evolution and Reengineering*, pages 538–548. IEEE, 2022.

[2] Aline Brito, Marco Tulio Valente, Laerte Xavier, and Andre Hora. You broke my code: Understanding the motivations for breaking changes in APIs. *Empirical Software Engineering*, 25:1458–1492, 2020.

[3] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. Apidiff: Detecting api breaking changes. In *International Conference on Software Analysis, Evolution and Reengineering*, pages 507–511. IEEE, 2018.

[4] Barthelemy Dagenais and Martin P Robillard. Semdiff: Analysis and recommendation support for api evolution. In *International Conference on Software Engineering*, pages 599–602. IEEE, 2009.

[5] Xingliang Du and Jun Ma. Aexpy: Detecting API breaking changes in python packages. In *International Symposium on Software Reliability Engineering*, pages 470–481. IEEE, 2022.

[6] Stéphane Ducasse, Guillermo Polito, Oleksandr Zaitsev, Marcus Denker, and Pablo Tesone. Deprewriter: On the fly rewriting method deprecations. *The Journal of Object Technology*, 21(1):1–23, 2022.

[7] Andre Hora, Romain Robbes, Marco Tulio Valente, Nicolas Anquetil, Anne Etien, and Stephane Ducasse. How do developers react to API evolution? a large-scale empirical study. *Software Quality Journal*, 26(1):161–191, 2018.

[8] André Hora and Marco Tulio Valente. apiwave: Keeping track of api popularity and migration. In *International Conference on Software Maintenance and Evolution*, pages 321–323. IEEE, 2015.

[9] Kaifeng Huang, Bihuan Chen, Linghao Pan, Shuai Wu, and Xin Peng. Repfinder: Finding replacements for missing apis in library update. In *International Conference on Automated Software Engineering*, pages 266–278. IEEE, 2021.

[10] Ricardo Job and Andre Hora. Availability and usage of platform-specific apis: A first empirical study. In *International Conference on Mining Software Repositories*, pages 27–31. IEEE, 2024.

[11] Ricardo Job and Andre Hora. How and why developers implement OS-specific tests. *Empirical Software Engineering*, 30(1):1–33, 2025.

[12] Ricardo Job and Andre Hora. On the implementation of os-specific tests: The cpython case. In *Brazilian Symposium on Systematic and Automated Software Testing*, pages 46–54, 2025.

[13] Ricardo Job and Andre Hora. Platform-specific apis, November, 2025.

[14] Dino Konstantopoulos, John Marien, Mike Pinkerton, and Eric Braude. Best principles in the design of shared software. In *International Computer Software and Applications Conference*, pages 287–292, 2009.

[15] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? an empirical study on the impact of security advisories on library migration. *Empirical Software Engineering*, 23:384–417, 2018.

[16] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. API change and fault proneness: A threat to the success of android apps. In *Joint Meeting on Foundations of Software Engineering*, pages 477–487, 2013.

[17] Tarek Mahmud, Meiru Che, and Guowei Yang. Android api field evolution and its induced compatibility issues. In *International Symposium on Empirical Software Engineering and Measurement*, pages 34–44, 2022.

[18] Matias Martinez and Bruno Gois Mateus. How and Why did developers migrate Android Applications from Java to Kotlin? A study based on code analysis and interviews with developers. *arXiv preprint arXiv:2003.12730*, 2020.

[19] Gabriel Menezes, Bruno Cafeo, and Andre Hora. How are framework code samples maintained and used by developers? the case of android and spring boot. *Journal of Systems and Software*, 1:1–30, 2021.

[20] Simon Moser and Oscar Nierstrasz. The effect of object-oriented frameworks on developer productivity. *Computer*, 29(9), 1996.

[21] Romulo Nascimento, Andre Hora, and Eduardo Figueiredo. Exploring API Deprecation Evolution in JavaScript. In *International Conference on Software Analysis, Evolution and Reengineering*, pages 169–173. IEEE, March 2022.

[22] Steven Raemaekers, Arie van Deursen, and Joost Visser. Measuring software library stability through historical version analysis. In *International Conference on Software Maintenance*, pages 378–387, 2012.

[23] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated API Property Inference Techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, May 2013.

[24] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. On the reaction to deprecation of clients of 4 + 1 popular Java APIs and the JDK. *Empirical Software Engineering*, 23(4):2158–2197, 2018.

[25] The Python Standard Library. https://docs.python.org/3/library/index.html, November, 2025.

[26] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. Historical and impact analysis of api breaking changes: A large-scale study. In *International Conference on Software Analysis, Evolution and Reengineering*, pages 138–147. IEEE, 2017.

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