The Python/C API: Evolution, Usage Statistics, and Bug Patterns

Mingzhe Hu and Yu Zhang
Lab for Intelligent Networking and Knowledge Engineering (LINKE)
University of Science and Technology of China, Hefei, China
Email: hmz18@mail.ustc.edu.cn, yuzhang@ustc.edu.cn

***Abstract*—Python has become one of the most popular programming languages in the era of data science and machine learning, especially for its diverse libraries and extension modules. Python front-end with C/C++ native implementation achieves both productivity and performance, almost becoming the standard structure for many mainstream software systems. However, feature discrepancies between two languages can pose many security hazards in the interface layer using the Python/C API. In this paper, we applied static analysis to reveal the evolution and usage statistics of the Python/C API, and provided a summary and classification of its 10 bug patterns with empirical bug instances from Pillow, a widely used Python imaging library. Our toolchain can be easily extended to access different types of syntactic bug-finding checkers. And our systematical taxonomy to classify bugs can guide the construction of more highly automated and high-precision bug-finding tools.**

***Index Terms*—Python/C API, Static analysis, Evolution analysis, Fact extraction, Bug pattern**

I. Introduction

Many real-world software systems are multilingual, which consist of components developed in different programming languages. Multilingual software systems can reuse code and combine language advantages. To support this, most programming languages provide Foreign Function Interface (FFI) to interact with native methods. This brings convenience to developers and is common in engineering practice. However, due to discrepancies between language features, *e.g.,* memory management [1, 2], exception handling [3, 4], and type system [5, 6], it is not easy to write secure and reliable multilingual programs.

As a popular programming language, Python is widely used in many fields, including data analysis, system administration, machine learning, etc. Python has a powerful ecosystem and community, diverse libraries and extension modules. Combined with its flexible syntax, Python provides developers with great programming convenience and rapid prototyping capability. On the other hand, Python receives intense criticism for its performance issue, which can also be made up by implementing critical parts in C/C++ extensions. In the wave of data science and deep learning, many mainstream frameworks choose Python as the default front-end language, and use the Python/C API to interact with C/C++ methods. This design is adopted by more and more software systems. However, due to inadequate understanding of the Python/C API, complexity of the interface itself and some design issues, interface code faces many potential security hazards.

```
1  // Pillow/src/encode.c
2  static ImagingEncoderObject* PyImaging_EncoderNew(int
       contextsize) {
3      ...
4      encoder = PyObject_New(ImagingEncoderObject,
5                             &ImagingEncoderType);
6      ...
7  }
8  PyObject* PyImaging_Jpeg2KEncoderNew(...) {
9      encoder = PyImaging_EncoderNew(
10                 sizeof(JPEG2KENCODESTATE));
11     ...
12     if (...) {
13         PyErr_SetString(PyExc_ValueError, "...");
14         Py_DECREF(encoder);
15         return NULL;
16     }
17 }
```

Fig. 1. A bug of mishandling exceptions

Fig. 1 shows an example of using the Python/C API in a C extension module of Pillow [7], a widely used Python imaging library. Variable `encoder` is allocated using the Python/C API `PyObject_New`, and `PyErr_SetString` sets error message when an error occurs. `PyImaging_EncoderNew` is a user-defined function named with prefix `Py`, following the Python/C API standard. After catching an exception, extension module must explicitly terminate the native method and return control to the Python interpreter. Before this, it is necessary to release the occupied resources such as the heap space of `encoder`. However, such handling (lines 14-15) is often missed by developers because they need not do this in pure Python code.

In this paper, we analyzed the scale, ~~evolution~~ and usage statistics of the Python/C API through static analysis. At the same time, we systematically analyzed and summarized the security risks of Python/C API for the first time, and proposed 10 classes of bug patterns including common cross-language bugs and their behaviors in the Python/C situation, and some Python-specific ones. We used some empirical bug instances collected from Pillow to illustrate these bug patterns.

Python continues to be popular, but there is little research on the interface security of its FFI. We hope our work will attract more academic and industrial attention on this growing topic. Our early research makes the following contributions:

- We implemented a toolchain *P*yCEAC (**Py**thon/**C** API **E**xtraction **A**nalysis and Bug **C**heckers), by applying it we obtained the evolution of Python/C API among big

versions since Python 2.7.0, and usage ~~statisticals~~ in seven popular software systems from different domains.
- We proposed 10 classes of bug patterns illustrated with empirical bug instances, and this is the first systematic summary of bug patterns concerning the Python/C API. Some bug patterns such as memory management flaws and buffer/integer overflow are traditional but different in multilingual scenarios. Others such as mishandling exceptions, insufficient error checking, Python/C API evolution flaws, reference counting errors, type misuses and GIL flaws are specific to Python/C interoperability.

II. Related Work

Recent research on FFI security of programming languages has mainly focused on Java and its FFI – Java Native Interface (JNI). Tan *et al.* conducted an empirical security study of native code in the JDK [8], and proposed some bug patterns. While for language discrepancies like dynamic typing, interpreted execution, memory system, interface design, etc., it can be quite different in the situation of Python/C.

Li *et al.* studied the problems related to exception handling in the JNI [3]. Exceptions thrown by JNI do not immediately return control to the JVM. They implemented a static analysis framework called JET [4] for exception and taint analysis.

Furr *et al.* carried out some research on FFI and type system. They proposed a type inference system for checking the OCaml/C interface, which could avoid type and memory security problems in native methods [5]. They further extended it to support polymorphic type inference of the JNI [6].

The Python/C API security-related research is rare and focuses on the classic bug patterns and reference counting errors. Python uses reference counting to manage its heap objects, but extension modules written in C/C++ are outside Python's memory management system. Pungi [1] used affine abstraction to statically analyze the SSA form of programs. RID [2] used inconsistent path pair checking to relax Pungi's hypothesis and improved analysis accuracy.

There are empirical studies on Python source code change-proneness [9, 10]. They showed files with dynamic features are more change-prone, and implemented a tool called PyCT to extract and classify fine-grained source code changes.

Most tools mentioned above are not open sourced, and none of them conducts a comprehensive summary of the Python/C API bug patterns. Although our ultimate goal is to build highly automated tools, there is currently no general methodology to guide the bug identification. So the first important step is to collect empirical evidence and characterize bug patterns.

III. Extraction and Analysis of the Python/C API

In this section, we first describe the design of our toolchain *PyCEAC* on Python/C API extraction and analysis, then use *PyCEAC* to analyze the evolution of Python/C API and their usage statistics in mainstream open source projects.

A. Extraction and Analysis Modules in PyCEAC

PyCEAC includes two parts shown in Fig. 2. The left part is designed for parsing a given version i of CPython, which is the default Python implementation in C, to extract macro definitions M_i and function declarations A_i of the Python/C API. While the right part is designed for analyzing the Python/C API usage in a given Python/C project j.

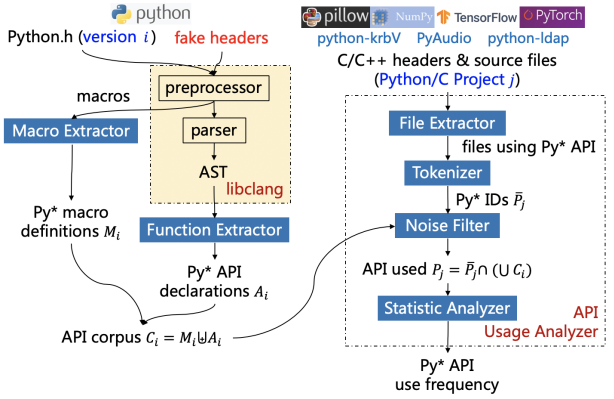

Fig. 2. Overview of toolchain *PyCEAC*: Python/C API extraction and analysis

On the left side of Fig. 2, both *Macro Extractor* and *Function Extractor* first call **libclang** library to access Clang front-end components [11] for parsing the header file `Python.h` and its included header files in CPython. Then *Macro Extractor* gets the results from the **preprocessor**, and further extracts all definitions of macros prefixed with `Py`; while *Function Extractor* constructs an Abstract Syntax Tree (AST) visitor based on **libclang** to access the AST obtained from the **parser** and to get all `Py*` API function declarations. All Python/C API corpus C_i is a disjoint union of A_i and M_i.

To handle non-C-standard types and platform/version-specific definitions when parsing the `#defines` and the `typedefs` in function prototype extraction, we borrowed the idea of "fake headers" from **pycparser** [12] by establishing some fake header files containing parts of the original.

On the right side of Fig. 2, *API Usage Analyzer* consists of four main components. Firstly, *File Extractor* separates files using the Python/C API from C/C++ source code in Project j. Secondly, *Tokenizer* is built atop **libclang** to extract IDs prefixed with `Py` from the separated files, the result is denoted as $\bar{P}_j$. Thirdly, *Noise Filter* uses set operations to remove user-defined functions whose names start with `Py` but are not in the obtained Python/C API corpus, and obtains $P_j = \bar{P}_j \cap (\bigcup_i C_i)$. Finally, *Statistic Analyzer* calculates usage statistics such as frequency and hotspot of the Python/C API in each project.

B. Evolution of the Python/C API

With *PyCEAC* we developed, we extracted the Python/C API defined/declared in each big release version of CPython since 2.7.0. TABLE I lists the comparison results which reflect the evolution of Python/C API among different versions.

The first two columns in TABLE I list the version number and release time of each analyzed CPython version. Column 3

TABLE I
EVOLUTION OF THE PYTHON/C API

Version	Date	Usage	API	Add	Remove	Change	Macro
2.7.0	2010.7.3	14.9%	563	-	-	-	252
3.2.0	2011.2.20	-	663	179	79	30	322
3.3.0	2012.9.29	-	702	115	76	2	274
3.4.0	2014.3.17	2.5%	732	30	0	16	276
3.5.0	2015.9.13	9.2%	751	19	0	2	288
3.6.0	2016.12.23	45.4%	764	13	0	9	292
3.7.0	2018.6.27	25.5%	804	43	3	6	298

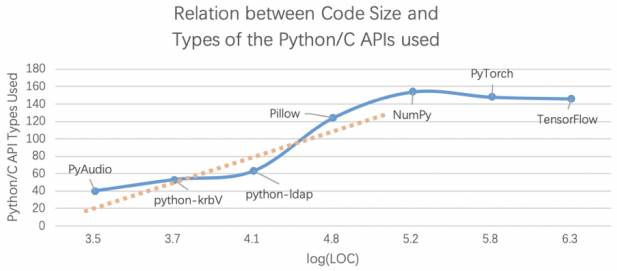

Fig. 3. Relation between code size and types of the Python/C APIs used

reflects developer usage proportion of the version, which comes from Python developer survey 2018 [13]. We extracted user-visible Python/C API functions in each version using *Function Extractor*, and further counted the total number and the number of add/remove/change, listed in Columns 4 to 7 of the table. The last column lists the number of macros prefixed with `Py` obtained from *Macro Extractor*.

RQ1: How is the evolution of Python/C API among Python versions? From TABLE I, we see that there are nearly 1000 Python/C API functions and macros in each version, and the total number grows with version updates, indicating that the Python/C API is continuously enriched to support more C extensions. Furthermore, the remove and change numbers, which are dangerous for binary compatibility, slow down since version 3.4.0. This shows that the existing functions tend to be stable. In addition, more than 80% of developers use versions after 3.4.0, so binary compatibility is no longer a big concern.

C. Python/C API Usage Statistics in Software Systems

In order to understand usage statistics of the Python/C API in real-world software, we choose mainstream open source projects with different application domains and different code sizes, of which C/C++ code sizes are more than 20% of total lines of code (LoC).

TABLE II lists the basic information and usage statistics of seven open source projects, where the two rightmost columns give the distinct Python/C API callee numbers and the total number of Python/C API callsites, which are collected using our *API Usage Analyzer* in *PyCEAC*.

RQ2: What is the usage statistics of Python/C API in mainstream open source projects? From TABLE II, we see that code size of a project is generally a positive relation to the type of Python/C API used. But when code reaches a certain size, the API types used will no longer grow and stop around 150, far less than the total API types (see Fig. 3). This also shows that the interface layer is glue code with specific functionality. Its complexity as well as bug patterns will not continue to grow as the code size increases. Understanding the Python/C API subset actually used allows us to sharply reduce the checking range of bug-finding tools.

IV. TAXONOMY TO CLASSIFY BUGS

Having evolution and usage statistics of the Python/C API, we next focus on **RQ3: What are the bug patterns concerning the Python/C API?** We intend to cover as many representative classes of bug patterns as we can. Some bug patterns are derived from other multilingual research discussed in Section II, *e.g.,* mishandling exceptions (IV-A), TOCTOU (IV-F2), and reference counting errors (IV-F3). Some are analyzed from official Python/C API manual [14], *e.g.,* insufficient error checking (IV-B), integer/buffer overflow (IV-C/IV-F1), and Python/C API evolution flaws (IV-E). Some are derived from language features, *e.g.,* memory management flaws (IV-D), GIL flaws (IV-F5), and type misuses (IV-F4). For each bug pattern, we manually searched and analyzed the issue lists of projects in TABLE II, illustrated with pull requests [15–18].

A. Mishandling Exceptions

When an exception is raised in pure Python code, Python interpreter will terminate it immediately and pass the control to the nearest `try` statement and match the exception type for exception handling. However, due to discrepancy of exception handling mechanism between the Python/C API and Python itself, exception raised using the Python/C API cannot immediately terminate the execution of native method. Therefore, there must be explicit return to avoid unexpected control flows. Before this, it is also necessary to properly handle the reference counts of allocated objects. Fig. 1 shows a bug from Pillow, where code at lines 14 and 15 should be added.

B. Insufficient Error Checking

Besides some Python/C APIs explicitly throwing exceptions, other APIs identify internal errors by return values, which should be checked before subsequent operations on relevant objects. The Python/C APIs generally use `NULL` or `-1` to identify an error, depending on the type of return value. But a few APIs such as `PyObject_HasAttr` always execute successfully, and very a few APIs such as `PyArg_Parse*` family use 0 for errors. Fig. 4(a) shows a bug where code at line 3 should be added to check the returned `item`.

C. Integer Overflow

Python/C APIs like `PyArg_Parse*` use format string to represent type conversions in the interface code, which actually has potential integer overflow errors. Unsafe format characters and their meanings are listed in TABLE III. Based on the table, we found a bug instance shown in Fig. 4(b).

TABLE II
THE PYTHON/C API IN SOFTWARE SYSTEMS

Project	Version	Description	Code Size (KLoC)	C/C++ Code Ratio	API	Frequency
Pillow	5.4.1	image processing	65.4	40.8%	124	1047
NumPy	v1.1.0rc1	scientific computing	151.9	55.5%	154	5864
TensorFlow	v1.10.0	machine learning	2185.1	51.6%	146	1194
PyTorch	v1.0rc1	machine learning	690.2	57.9%	148	1205
python-krbV	1.0.90	web authentication protocal	4.9	86.4%	53	736
PyAudio	0.2.11	audio I/O	3.3	57.2%	40	339
python-ldap	3.2.0	directory access protocol	12.6	22.5%	63	300

```
// Pillow/src/_imaging.c#L2004
PyObject* item = Py_BuildValue("iN", v->count,
     getpixel(self->image, self->access, v->x,
      v->y));
if (item == NULL) {...}
PyList_SetItem(out, i, item);
```

(a) A bug of insufficient error checking

```
// Pillow/src/encode.c#L997
if (!PyArg_ParseTuple(args, "ss|OOOsOIOOOssi", &mode, &format,
      ...))
    return NULL;
```

(b) A bug of integer overflow

```
// Pillow/src/libImaging/Resample.c#L622
ksize_horiz = precompute_coeffs(imIn->xsize,
      box[0], box[2], xsize, filterp,
      &bounds_horiz, &kk_horiz);
if ( ! ksize_horiz) { return NULL; }

ksize_vert = precompute_coeffs(imIn->ysize,
    box[1], box[3], ysize, filterp,
     &bounds_vert, &kk_vert);
if ( ! ksize_vert) {
    free(bounds_horiz);  free(kk_horiz);
    free(bounds_vert); free(kk_vert);
    return NULL;
}
```

(c) Bugs related to memory management

```
// Pillow/src/_webp.c#L851
#if PY_VERSION_HEX >= 0x03000000
PyMODINIT_FUNC PyInit__webp(void) {
    ...    m = PyModule_Create(&module_def);  ...
}
#else
PyMODINIT_FUNC init_webp(void) {
    PyObject* m = Py_InitModule("_webp", webpMethods); ...
}
#endif
```

(d) A false positive of the API evolution flaws

```
// Pillow/src/encode.c#L128
int bufsize = 16384;
if (!PyArg_ParseTuple(args, "|i", &bufsize))    return NULL;
```

(e) A bug related to Python/C API evolution

Fig. 4. Empirical code snippets

TABLE III
FORMAT CHARACTERS WITHOUT OVERFLOW CHECKING

Format Character	Python Type	C Type
B	integer	unsigned char
H	integer	unsigned short int
I	integer	unsigned int
K	integer	unsigned long long
k	integer	unsigned long

TABLE IV
FOUR FAMILIES OF MEMORY ALLOCATORS

Memory Allocator	Since	Default Versions	Thread Safety	Binary Compatibility	Small Object Optimization
malloc	-	< 3.4	Y	Y	N
PyMem_RawMalloc	3.4	3.4, 3.5	Y	Y	N
PyMem_Malloc	3.4	≥ 3.6	N	Y	Y
PyMem_MALLOC	3.4	-	N	N	Y

D. Dynamic Memory Management Flaws

Python allocates objects on a private heap, and the Python/C API interface layer supports four families of memory allocators listed in TABLE IV. Within each family, there can be three traditional dynamic memory management errors: 1) memory leaks; 2) multiple frees; 3) dereference dangling pointers. Moreover, there are also mismatches in the same or different families of memory allocators. For example, memory allocated via `PyMem_Malloc` should be reclaimed by `PyMem_Free`, while using `PyMem_Del` or `free` are both wrong. Fig. 4(c) shows two memory bug instances, where entering the second `if` conditional means that `*_horiz` have been allocated, while `*_vert` have not and should not be freed.

E. Python/C API Evolution Flaws

Python has an active community constantly updating the language itself. Python version series 2 and 3 are not fully compatible [19]. The Python/C API is not binary compatible (ABI), adding or changing fields may break the ABI, users have to recompile extensions for different Python versions.

Fig. 4(d) shows an example that mainstream applications often use conditional compilation to handle different versions. For developers who lack multilingual programming experience or multi-version tests, this handling is often overlooked, which can bring tons of failure feedback when open sourced.

In addition, type declaration is a notable change. When indexing Python sequence, `Py_ssize_t` should be used instead of C `int`. The former has the same length as the compiler's `size_t`, but is signed. While with C `int`, sequences have at

most 2^{31} elements on 64-bit machines [20]. This bug pattern is closely related to format characters. Character `i` converts Python integer to C `int`, while character `n` to `Py_sszie_t`. Fig. 4(e) shows a bug instance.

F. Other Bug Patterns

We next mention the other five classes of bug patterns without empirical bug instances, for that the first two are easy to trigger, and bug-finding for the other three requires careful design. We detail their behaviors here.

1) Buffer Overflow: Python performs automatic boundary checking on arrays, but values passed through the Python/C API may trigger buffer overflows in the interface layer. Also, the index can be negative on the Python side, which is an inappropriate behavior in the native code.

2) Time-of-check-to-time-of-use (TOCTOU): TOCTOU is a typical error caused by race conditions in file access [21]. Performing two or more file operations on a single file name or path name creates a race window between the two file operations. During that period of time, an attacker can alter the file or change the file link, potentially endangering safety.

3) Reference Counting Errors: Python uses reference counting to manage allocated objects [22]. Each Python object has an `ob_refcnt` field. Unfortunately, native object is not under the control of Garbage Collector (GC). When manipulating a Python object through the Python/C API, its reference count cannot be adjusted automatically, but has to be adjusted by explicitly calling the Python/C APIs like `Py_INCREF` and `Py_DECREF`. Considering the special cases of borrowing and stealing [23], this task is very error-prone. Actually, in Fig. 1, the `Py_DECREF` operation at line 14 is often overlooked.

4) Type Misuses: Host languages with dynamic typing increase the risk of misusing types. The Python/C API contains some functions for type conversion, *e.g.,* `PyLong_AsLong`, which represents the passed Python integer as a C `long`. Since Python is not statically typed, type checking of the passed value can be a problem itself. Passing a value of unmatched type will cause an error or some unexpected behavior.

5) Bugs Related to GIL: Python interpreter is not fully thread-safe. Current thread must hold the Global Interpreter Lock (GIL) to securely manipulate Python objects. When creating a thread in Python, thread state is automatically associated with the thread. But when a thread is created by C, it has neither GIL nor thread state information. Thus some Python/C APIs must be called after holding the GIL.

V. Conclusion and Future Work

For multilingual software consisting of Python and C/C++ extension modules, security risks of the Python/C API are like bombs inside. In this paper, we reveal the evolution and usage statistics of the Python/C API, analyze and summarize the bug patterns. Our static analysis toolchain *PyCEAC* provides extraction and usage analysis of the Python/C API, and can easily add bug checkers based on syntactic pattern matching.

More and more mainstream software adopts multilingual architecture, making us believe that the realization of more practical tools, the improvement of basic design, and the static analysis of dynamic languages are worth further exploration. In addition to the trade-off between performance and productivity, Python safety and security deserve more attention.

Acknowledgment

This work was partially funded by the National Natural Science Foundation of China (No. 61772487) and Anhui Provincial Natural Science Foundation (No.1808085MF198).

References

[1] S. Li and G. Tan, "Finding reference-counting errors in Python/C programs with affine analysis," in *28th European Conference on Object-Oriented Programming (ECOOP)*, 2014, pp. 80–104.

[2] J. Mao, Y. Chen, Q. Xiao, and Y. Shi, "RID: Finding reference count bugs with inconsistent path pair checking," in *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016, pp. 531–544.

[3] S. Li and G. Tan, "Finding bugs in exceptional situations of JNI programs," in *16th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2009, pp. 442–452.

[4] ——, "JET: Exception checking in the Java native interface," in *26th ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2011, pp. 345–358.

[5] M. Furr and J. S. Foster, "Checking type safety of foreign function calls," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 62–72.

[6] ——, "Polymorphic type inference for the JNI," in *15th European Conference on Programming Languages and Systems (ESOP)*, 2006, pp. 309–324.

[7] Pillow. [Online]. Available: https://python-pillow.org

[8] G. Tan and J. Croft, "An empirical security study of the native code in the JDK," in *17th USENIX Conference on Security Symposium (USENIX Security)*, 2008, pp. 365–377.

[9] B. Wang, L. Chen, W. Ma, Z. Chen, and B. Xu, "An empirical study on the impact of Python dynamic features on change-proneness." in *27th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2015, pp. 134–139.

[10] W. Lin, Z. Chen, W. Ma, L. Chen, L. Xu, and B. Xu, "An empirical study on the characteristics of Python fine-grained source code change types," in *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 188–199.

[11] libclang: C interface to Clang. [Online]. Available: https://clang.llvm.org/doxygen/group__CINDEX.html

[12] On parsing C, type declarations and fake headers. [Online]. Available: https://eli.thegreenplace.net/2015/on-parsing-c-type-declarations-and-fake-headers/

[13] Python developers survey 2018 results. [Online]. Available: https://www.jetbrains.com/research/python-developers-survey-2018/

[14] Python/C API reference manual. [Online]. Available: https://docs.python.org/3/c-api/index.html

[15] Return after error. [Online]. Available: https://github.com/python-pillow/Pillow/pull/3967

[16] Pass the correct types to pyarg_parsetuple. [Online]. Available: https://github.com/python-pillow/Pillow/pull/3880

[17] Fixed freeing unallocated pointer when resizing with height too large. [Online]. Available: https://github.com/python-pillow/Pillow/pull/4116

[18] Fixed deprecation warnings. [Online]. Available: https://github.com/python-pillow/Pillow/pull/3749

[19] B. A. Malloy and J. F. Power, "Quantifying the transition from Python 2 to 3: An empirical study of Python applications," in *11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017, pp. 314–323.

[20] PEP 353 – using ssize_t as the index type. [Online]. Available: https://www.python.org/dev/peps/pep-0353/

[21] M. Bishop, M. Dilger *et al.*, "Checking for race conditions in file accesses," *Computing systems*, vol. 2, no. 2, pp. 131–152, 1996.

[22] T. W. Christopher, "Reference count garbage collection," *Software: Practice and Experience*, vol. 14, no. 6, pp. 503–507, 1984.

[23] A. J. H. Simons, "Borrow, copy or steal?: Loans and larceny in the orthodox canonical form," in *13th SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1998, pp. 65–83.