On the Security of Python Virtual Machines: An Empirical Study

Xinrong Lin Baojian Hua* Qiliang Fan
School of Software Engineering
University of Science and Technology of China, China
{lxr1210, sa613162}@mail.ustc.edu.cn bjhua@ustc.edu.cn*

Abstract—**Python continues to be one of the most popular programming languages and has been used in many safety-critical fields such as medical treatment, autonomous driving systems, and data science. These fields put forward higher security requirements to Python ecosystems. However, existing studies on machine learning systems in Python concentrate on data security, model security and model privacy, and just assume the underlying Python virtual machines (PVMs) are secure and trustworthy. Unfortunately, whether such an assumption really holds is still unknown.**

This paper presents, to the best of our knowledge, the *first* and most *comprehensive* empirical study on the security of CPython, the official and most deployed Python virtual machine. To this end, we first designed and implemented a software prototype dubbed PVMSCAN, then use it to scan the source code of the latest CPython (version 3.10) and other 10 versions (3.0 to 3.9), which consists of 3,838,606 lines of source code. Empirical results give relevant findings and insights towards the security of Python virtual machines, such as: 1) CPython virtual machines are still vulnerable, for example, PVMSCAN detected 239 vulnerabilities in version 3.10, including 55 null dereferences, 86 uninitialized variables and 98 dead stores; Python/C API-related vulnerabilities are very common and have become one of the most severe threats to the security of PVMs: for example, 70 Python/C API-related vulnerabilities are identified in CPython 3.10; 3) the overall quality of the code remained stable during the evolution of Python VMs with vulnerabilities per thousand line (VPTL) to be 0.50; and 4) automatic vulnerability rectification is effective: 166 out of 239 (69.46%) vulnerabilities can be rectified by a simple yet effective syntax-directed heuristics.

We have reported our empirical results to the developers of CPython, and they have acknowledged us and already confirmed and fixed 2 bugs (as of this writing) while others are still being analyzed. This study not only demonstrates the effectiveness of our approach, but also highlights the need to improve the reliability of infrastructures like Python virtual machines by leveraging state-of-the-art security techniques and tools.

Index Terms—**Empirical, Python virtual machines, Security**

I. INTRODUCTION

Python continues to be one of the most popular and important programming languages in the era of data science. According to the latest survey from IEEE [1], TIOBE [2] and Stack Overflow [3], Python is ranked as the top language in 2021. Moreover, the Python developer survey [9] reports, over 85% of the survey respondents use Python as their main programming language. As a result, Python has been widely used in many application fields, including data science [11] [12], Web development [13] [14], machine learning [15] [16], medical treatment [17] [18], autonomous driving systems [19] [20], etc. Many fields (e.g., medical treatment or autonomous driving systems) are safety-critical, they put forward higher security requirements to the whole Python ecosystem.

There has been a significant amount of research on the security, reliability, and trustworthy on the machine learning systems in Python with respect to data security, model security and model privacy [4] [5] [6] [7] [8]. Unfortunately, there is little research on the security of the underlying Python virtual machines (PVMs); instead, existing studies just assume these PVMs are secure. However, whether such an assumption really holds is still unknown.

One may speculate that the study of PVMs security is a solved problem, as there has been a significant amount of research in this direction [28] [64] [65] [67] [68]. However, there are 3 challenges remaining: first, the implementation language of PVMs may be vulnerable thus defeating the security guarantees of PVM. For example, CPython, the official and most deployed PVM, consists of more than 400,000 lines of C code. Due to the notorious insecure nature of C, many CVEs (e.g., [22] [23] [24] [25]) are reported recently. We argue that, just as the recent Log4j vulnerability demonstrated [104] (CVE-2021-44832), even a single vulnerability in infrastructures such as PVMs will lead to serious consequences.

Second, PVMs have some unique properties which makes it difficult to detect vulnerabilities in them. To be specific, PVMs make use of 831 Python/C APIs [10], which allow Python programs to interoperate with programs in C/C++. These Python/C APIs have complex security requirements, due to the semantics discrepancy between Python and C. As a result, insufficient checking of the Python/C APIs may lead to vulnerabilities, such as insufficient error checking, reference counting error [69] [70] [71], type misuses, and dynamic memory management [26] [62], etc. Unfortunately, to the best of our knowledge, no state-of-the-art scanning engines can detect such Python/C API-related vulnerabilities.

Finally, it is challenging to rectify vulnerabilities in PVMs timely and automatically. On one hand, it is error-prone and time-consuming to rectify vulnerabilities manually. On the other hand, existing studies on automatic program rectification [75] [77] focus mostly on functional rectifications rather than

*Corresponding author.

security ones [74]. Worse yet, existing program rectification techniques are test-driven [72] [73], by measuring the quality of the rectification in terms of number of test cases passed. It is difficult to scale such techniques to realistic PVMs without comprehensive security test suits.

To this end, to study the security of PVMs, several key questions remain unanswered: are latest PVMs vulnerable? How do vulnerabilities evolve in different versions of PVMs? Are Python/C API-related vulnerabilities common in current PVMs? How difficult and costly is it to rectify timely and automatically vulnerabilities in PVMs? Without such knowledge, PVM developers cannot benefit from the state-of-the-art security tools, tool builders might be building on wrong assumptions and researchers might miss opportunities for improving the state of the art.

Our work. To answer the aforementioned key questions, this paper presents, to the best of our knowledge, the *first* and most *comprehensive* empirical study of security of CPython, the official and most deployed virtual machine. This study is performed in several steps. First, to detect vulnerabilities in PVMs, we designed and implemented a novel software prototype dubbed PVMSCAN, by leveraging off-the-shelf state-of-the-art vulnerability analysis tools.

Second, to investigate vulnerabilities in PVMs, we first applied PVMSCAN to CPython 3.10, the latest version consisting of 479,044 lines of source code. And to study vulnerabilities evolution, we applied PVMSCAN to *all* 11 versions of CPython (3.0-3.10) with a total of 3,838,606 lines of code.

Finally, to study automatic vulnerability rectification, we utilize a simple yet effective syntax-directed heuristics to insert minimally intrusive security patches.

Empirical results give interesting findings and insights, such as: 1) the latest CPython VM contains considerable vulnerabilities: we have identified 239 vulnerabilities (version 3.10), including 55 null dereferences, 86 uninitialized variables and 98 dead stores; 2) Python/C API-related vulnerabilities are very common and have become one of the most severe threats to the security of PVMs: we identified 70 Python/C API-related vulnerabilities in CPython 3.10; 3) the overall quality of the code remained relatively stable during the evolution of Python VMs: although the CPython grew significantly, from 248,203 lines of code in 3.0 to 479,044 lines of code in 3.10, the vulnerabilities per thousand lines (VPTL) grew slowly from 0.38 to 0.50; and 4) automatic vulnerability rectification is effective: 166 out of 239 (69.46%) vulnerabilities can be rectified by simple, yet effective syntax-directed heuristics.

Last but not least, PVMSCAN is efficient in vulnerability rectification: it takes 443.22 milliseconds to fix 166 vulnerabilities (2.67 millisecond per vulnerability) for the latest CPython VM version 3.10.

We have reported our initial empirical results to the CPython development team, and they acknowledged us and already confirmed and fixed 2 of these vulnerabilities (as of this writing) while other are still being analyzed.

Contributions. To the best of our knowledge, this work represents the first step towards a comprehensive empirical study of Python virtual machine security. To summarize, our work makes the following contributions:

- **Empirical study and tools.** We presented the first and most comprehensive empirical study on the security of Python virtual machines, with a novel software prototype we created dubbed PVMSCAN.
- **Findings and insights.** We presented empirical results, findings from the analysis, as well as implications for these results, future challenges and research opportunities.
- **Open source.** We make our tool and empirical data publicly available in the interest of open science at `https://doi.org/10.5281/zenodo.6421694`.

Outline. The rest of this paper is organized as follows. Section II presents the background for this work. Section III presents the methodology for the study. Section IV presents empirical results, by answering research questions. Section V and VI discusses implications for this work, and threats to validity, respectively. Section VII discusses the related work, and Section VIII concludes.

II. BACKGROUND

To be self-contained, we present, in this section, necessary background information on Python virtual machines (II-A), and Python/C API (II-B).

A. Python Virtual Machines (PVM)

Python is a dynamically typed and interpreted language. Python programs are first compiled by Python compilers to an intermediate format called Python bytecode [66], which are then interpreted by underlying Python virtual machines. This design choice makes Python a portable language to run on any platforms with Python virtual machine implementations.

Python virtual machines are managed runtimes for Python, which not only interpret Python bytecode, but also manage low-level functionalities such as memory management, thread management, and foreign function invocations, etc.

CPython [21] is the official and most deployed PVM which has more than 21.7K forks according to its official statistics. CPython is a software with nontrivial code size, consisting of more than 479,044 lines of C code in 344 source files as presented by TABLE I (for the current stable version, 3.10).

TABLE I
CPYTHON 3.10 CODE STATISTICS

Module	#Files	LoC
Include	81	10,408
Modules	112	229,582
Objects	46	96,399
Parser	10	40,280
PC	19	10,611
Programs	3	2,071
Python	72	89,514
Total	344	479,044

CPython is still under active development and has grown significantly in code sizes. For example, from version 3.0 to 3.10, the number of source files grew from 315 to 344, and lines of source code grew from 248,203 to 479,044.

As CPython is implemented in C, it is vulnerable to security issues common to most C programs [64] [65] [67] [68], such as buffer overflows, integer overflows, heap overflows, stack overflows, and double free, etc. However, it should be noted that we are not criticizing the C language here. Indeed, C is an indispensable and de-facto system programming language to develop infrastructures such as Python virtual machines due to its extreme efficiency and flexibility. Unfortunately, C is also (arguably more) flexible in introducing subtle vulnerabilities. Worse yet, for important infrastructures like PVM, even a single bug can defeat its guarantee of security and lead to serious consequences, as demonstrated by the recent Log4j [104] exposure. To this end, it is important to guarantee the security and trustworthiness of Python virtual machines by leverage state-of-the-art security techniques and tools.

B. Python/C API

To develop multilingual applications interacting with external C/C++ code, Python introduced a Foreign Function Interface (FFI) called Python/C API [38]. The Python/C API is bidirectional: 1) it allows Python code to invoke native C/C++ code; and 2) it allows native C/C++ code to interact with the Python virtual machines, such as creating Python objects, manipulating Python objects, and performing garbage collection, etc.

It is challenging to use Python/C APIs correctly and securely, for two key reasons: first, the number of Python/C API are considerably large to support complex functionalities [62]. For example, the number of Python/C APIs has grown from 663 in version 3.2 to 831 in version 3.10 [101]. And the number of Python/C APIs is continuously growing with new PVM releases to add more functionality.

Second, Python/C APIs have tricky semantics due to discrepancies between Python and C. Prior research efforts [62] [69] have demonstrated that improper usage of these APIs can lead to subtle vulnerabilities such as mishandling exceptions, insufficient error checking, etc., which make the virtual machines vulnerable. For example, the following code snippet

```
1 // Python-3.10.0/Modules_heapqmodule.c
2 static int heapq_exec(PyObject *m){
3  PyObject *about = PyUnicode_FromString(..);
4 ++ if(NULL == about) error(...); // patch
5  if(PyModule_AddObject(m, "..", about) < 0){
6    Py_DECREF(about);
7    return -1;}
```

presents a sample Python/C API-related vulnerability detected and rectified by our tool PvmScan. The 3rd line invokes a specific Python/C API `PyUniode_FromString`, which, according to the API specification, may return nullable values; hence, subsequent accesses to the return value `about` (e.g., line 5 or 6) may trigger null dereference errors. A simple yet effective rectification (line 4) can eliminate such vulnerabilities.

Last but not least, we must point out that it is difficult to automatically check security of all these Python/C APIs, as they have much more complex security requirements and diverse security semantics than the above sample code shows. For example, the Python/C API specification specifies that the `Py_DECREF` API should be invoked before a function returns (line 6). Lack of this checking will lead to memory leaking.

III. Methodology

In this section, we present the methodology to conduct the empirical study. It is challenging to perform an empirical study for large projects like PVMs, for two key reasons: 1) automation: the study should be fully automated, otherwise it is difficult if not impossible to study large code base consisting of up to hundreds of thousands lines of source code in a fully automatic manner; human analysis is only required to complement the analysis by manual code inspection; and 2) scalability: the analysis should work for any PVMs with diverse structures instead of specific ones.

To this end, we designed and implemented a novel software prototype dubbed PvmScan to analyze and rectify vulnerabilities in PVMs, which is supplemented by human efforts to inspect code. We first discuss the architecture of PvmScan (III-A), then describe the design and implementation details of the frontend (III-B), the vulnerability analysis (III-C), the Python/C API purification (III-D), the automatic rectification (III-E), the validation (III-F), and the rectified program generation (III-G), respectively.

PvmScan has two key advantages: 1) it can leverage state-of-the-art vulnerability detection techniques and tools to improve security of PVMs; and 2) it is independent of the specific vulnerability detection techniques, which can range from static ones such as program analysis, to dynamic ones such as fuzzing.

A. The Architecture

Two principles guide the architecture design of PvmScan. First, the architecture of PvmScan should be extensible to support the analysis and rectification of different versions of same PVMs, or even different PVM implementations. Second, the architecture of PvmScan should be modular, hence each module can be easily extended or replaced separately.

Based on these design principles, we present the architecture of the PvmScan in Fig. 1, which consists of several key modules. First, the frontend module (❶) takes as input the corresponding PVM source files, processes the PVM source code, and outputs PVM native code. Second, the vulnerability analysis module (❷) takes as input the native code in the PVM, analyzes this code, and outputs a vulnerability report. Third, the purification module (❸) takes as inputs both the vulnerability report and the specification of Python/C API, purifies the report and generates a vulnerability summary. Forth, the automatic rectification module (❹) takes as inputs both the original PVM source code and the vulnerability summary, rectifies the source code of PVM according to the vulnerability summary, and generates patched PVM source code. Finally, the validation module (❺) takes as inputs the original PVM source code, the patched PVM code, and a

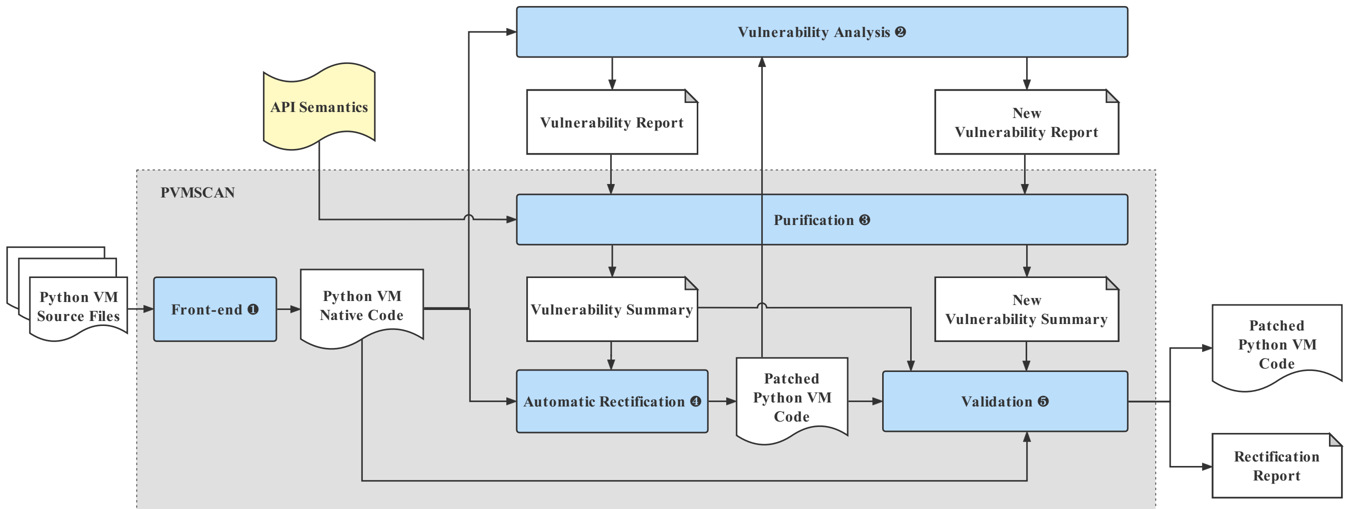

Fig. 1. The architecture of PVMSCAN

new vulnerability report generated by the vulnerability analysis module for the patched PVM source code, to validate the effect of rectification, and generates the final patched code, along with a rectification report.

In the following sections, we discuss the design and implementation details of each module, respectively.

B. The Frontend

Different Python virtual machines make use of different source organizations. Even for the same Python virtual machine, the source organization changes with software evolution. Furthermore, the sources contain scripts, tests, documents, besides source code. Some source code are automatically generated at building time.

To handle these peculiarities, the frontend normalizes the source code by: 1) filtering source code by removing irrelevant components that are not affected by security flaws, such as documents; 2) generating build-time sources; and 3) preprocessing C header files to speed up subsequent processing. Although combining the frontend with other phases is possible, the current design of PVMSCAN, from a software engineering perspective, has two key advantages: 1) it makes PVMSCAN feasible to process different versions of the same PVM, or even different PVMs; and 2) it is more efficient by filtering irrelevant code in an early stage.

C. Vulnerability Detection

To conduct an empirical study on the security of PVMs, we need to detect vulnerabilities in those PVMs. To this end, the vulnerability detection module of PVMSCAN leverages state-of-the-art and off-the-shelf vulnerability analysis engines, to generate a detailed vulnerability report for subsequent processing.

We have a key selection criterion for analysis engines: the selected engines should generate, for the detected vulnerabilities, complete and precise information. Such information includes but is not limited to vulnerability type, source files, line numbers, etc., as one of our research goals is to investigate to what extent the detected vulnerabilities can be rectified automatically (Section III-E). Without the detailed vulnerability information, it is difficult if not impossible to rectify it in an automated manner. For instance, it is difficult to rectify a vulnerability without source locations.

To this end, we have selected 8 popular state-of-the-art analysis engines as presented in TABLE II. All engines are actively maintained, with 4 of them updated this year whereas others were updated last year. All but two (Fortify and Coverity) are open source. Except for two proprietary softwares, 4 of the engines are licensed under GNU GPL.

TABLE II
ANALYSIS TOOLS USED IN THIS STUDY

Tools	Memory vulner-ability	Last updated	Open source	License
Infer [79]	✓	2022.03	✓	MIT
Clang [31]	✓	2021.11	✓	Apache v2.0
TscanCode [30]	✓	2022.01	✓	GNU GPL v3.0
Fortify [33]	✓	2022.01	✘	Proprietary
CppCheck [36]	✓	2022.02	✓	GNU GPL v3.0
Coverity [32]	✓	2021.12	✘	Proprietary
Valgrind [34]	✓	2021.10	✓	GNU GPL v2.0
FlawFinder [35]	✓	2021.07	✓	GNU GPL v2.0

This work mainly studies memory vulnerabilities, for two key reasons: 1) existing studies [62] have demonstrated such vulnerabilities are very serious and very common to PVMs; and 2) state-of-the-art security analysis tools are good at detecting such vulnerabilities (see TABLE II). To be specific, this study focuses on 4 type of memory vulnerabilities: null dereferences, dead stores, uninitialized variables, and resource/memory leaks. Nevertheless, it should be noted that our infrastructure PVMSCAN can be easily be extended to

detect other types of vulnerabilities as well.

D. Purification

The purification module takes as inputs both a vulnerability report generated by the vulnerability analysis engines along with a Python/C API semantics specification, and generates a vulnerability summary to be used in subsequent phases.

The purification module has one key functionality: it elaborates the original vulnerability report against the Python/C API semantics specification, to generate a separate Python/C API-related vulnerability report (although in the same vulnerability summary file).

The Python/C API elaboration module is indispensable because, although vulnerability analysis engines are effective in detecting general vulnerabilities, they lack the domain specific knowledge about Python/C API semantics. Such a lack of knowledge may lead to two problems: 1) false positives, which can lead to redundant rectifications; and 2) false negatives, which may lead to incorrect rectifications. For example, consider the following code snippet (from CPython version

```
1 // Python-3.10\_csv.c #line 37
2 static int _csv_clear(PyObject *module){
3  *module_state = PyModule_GetState(module);
4  Py_CLEAR(module_state->error_obj);
5  Py_CLEAR(module_state->dialects);}
```

3.10), for which some vulnerability analysis engines (e.g., Infer) incorrectly report null dereference vulnerabilities for the arguments `module_state` to the Python/C API `Py_CLEAR()`. However, the Python/C API specification specifies that `Py_CLEAR()` [102] accepts nullable arguments. Although add an extra nullable checking patch does not affect correctness, it does hurt performance by executing useless checking.

It should be noted that the presence of a purification module does not demonstrate the limitation of any analysis engines, as any programs analysis of nontrivial program properties on large software projects is conservative [97].

E. Automatic Rectification

The automatic rectification module takes as inputs the vulnerability summary from the purification module as well as the original PVM source code, automatically rectifies vulnerabilities and generates patched PVM code. The patched PVM code will be further processed by subsequent phases. It should be noted that the key research goal of this part of the study is to investigate to what extent the detected vulnerabilities can be rectified in an automated manner, thus giving deeper insights of the nature of PVM vulnerabilities. It does not intend to substitute developer code reviews or existing code quality testing infrastructures such as CI/DI.

Technically, existing techniques for program rectification [90] [91] [92] [93] [94] consist of either one type or a combination of two types of primitive operations: addition and removal of code segments. Our key insight here is that, to rectify security instead of functionality vulnerabilities, it is normally enough to add extra code segments without modifying existing ones, as vulnerabilities are often caused by lack of certain security checking such as releasing resources, freeing memory, or null checking [90]. As we will show (Section IV-I), this strategy is also used by the Python development team.

Based on this key insight, PvmScan employs a simple, yet effective syntax-directed heuristics. To be specific, our current proof-of-concept implementation supports automatic rectifications of the following 4 types of vulnerabilities (with more still being added): 1) null dereference: PvmScan adds guard statements [62], which performs runtime checking to guarantee nullable variables are not dereferenced; 2) uninitialized variables: PvmScan inserts necessary initialization statements, based on variable types; 3) dead stores: PvmScan removes the corresponding dead statements conservatively; and 4) resource leaking: PvmScan adds explicit reclaiming statements to reclaim resources or memories.

F. Validation

We have two primary goals for validations: 1) detected vulnerabilities have indeed been rectified; and 2) normal functionality of the target PVM is not affected.

To achieve the first goal, we leverage a differential testing technique, that is, the patched code is fed to vulnerability analysis engines for a second time, to obtain a new vulnerability summary. Next, the validation module compares the new summary against the original one, to check whether vulnerabilities disappear.

To achieve the second goal, we utilize the standard regression testing technique, to test the rectified PVM against the testing suits in the PVM distribution. Although regression is incomplete in theory, it is a well established practice for program testing, and our experimental results with PvmScan demonstrated this strategy is effective in practice.

Rectifications for a target vulnerability V are successful when two conditions hold simultaneously: 1) V disappears in the patched code; and 2) the functionality of PVM is not affected.

G. Rectified Program and Report Generation

After rectifying the target PVM, PvmScan generates as outputs the rectified programs, as well as a report for subsequent analysis.

IV. Empirical Results

In this section, we present the empirical results by answering research questions.

A. Research Questions

By presenting the empirical results, we mainly investigate the following research questions:

RQ1: Effectiveness. Is PvmScan effective in detecting vulnerabilities in Python virtual machines? How about false positives and false negatives of PvmScan?

RQ2: Python/C API related vulnerabilities. As PvmScan is proposed to detect Python/C API-related vulnerabilities in

TABLE III
VULNERABILITIES DETECTED BY PVMSCAN

Tools	ND		UV		DS		RL/ML		Total				
	Reports	TPs	Reports	TPs	Reports	TPs	Reports	TPs	Reports	TPs	Precision	Recall	F1
Infer	211	55	245	86	217	98	0	0	673	239	35.51%	41.07%	38.09%
Clang	34	9	11	4	608	259	0	0	653	272	41.65%	46.74%	44.05%
TscanCode	61	15	17	8	0	0	0	0	78	23	29.49%	3.95%	6.97%
Fortify	22	12	0	0	7	4	40	0	69	16	23.19%	2.75%	4.92%
CppCheck	10	6	43	18	0	0	0	0	53	24	45.28%	4.12%	7.56%
Coverity	32	8	0	0	0	0	3	0	35	8	22.86%	1.37%	2.59%
Valgrind	0	0	0	0	0	0	1	0	1	0	0.00%	0.00%	/
FlawFinder	0	0	0	0	0	0	0	0	0	0	/	0.00%	/
Total	370	105	316	116	832	361	44	0	1562	582	/	/	/

PVMs, is PVMSCAN effective in detecting Python/C API-related vulnerabilities?
RQ3: Evolution. How do the vulnerabilities evolve in different versions of PVMs?
RQ4: Rectification. Is PVMSCAN effective in rectifying vulnerabilities in CPython virtual machines automatically?
RQ5: Usefulness. Are empirical study and rectification results useful to the Python development team?

B. Experimental Setup

All the experiments and measurements are performed on a server with one 4 physical Intel i7 core (8 hyperthread) CPU and 12 GB of RAM running Ubuntu 21.04.

C. Data Sets

We selected 11 different versions of CPython virtual machines (from 3.0 to 3.10) as our empirical data sets. Two principles guided our selection of data sets: 1) we selected CPython, as it is the default, official and most widely deployed Python VM [78]; and 2) we only selected Python versions 3.x but not versions 2.x, because according to a latest Python developer survey [9], the use rate of Python 3.x is over 94.0% and growing. Moreover, the official Python team [78] has deprecated Python 2.x. However, it should be noted that PVMSCAN can be used to process Python 2.x legacy code without any technical difficulties.

D. Evaluation Metrics

We use precision and recall to measure the accuracy (or effectiveness) of our tool. The definition of these two metrics is in equation 1.

$$precision = \frac{tp}{tp + fp} \quad recall = \frac{tp}{tp + fn} \tag{1}$$

In the equation, we use tp, fp, fn to denote true positives, false positives and false negatives, respectively. We take the union of true positives from all analysis engines as our ground truth, and recall measures the ratio of true positives to the ground truth. An analysis engine with high false negatives may have low recall. Precision measures as the ratio of true positives to the result of an engine. An engine with high false positives may have low precision. Given the importance of both recall and precision, we also compute the $F1$ score according to equation 2.

$$F1\,score = \frac{2 \times precision \times recall}{precision + recall} \tag{2}$$

F1 score can reflect the overall accuracy of an analysis engine.

E. Effectiveness

To answer **RQ1** by investigating the effectiveness of PVMSCAN in detecting vulnerabilities, we first apply PVMSCAN to the latest CPython version 3.10. TABLE III presents the empirical results: the first column gives the names of the tools. The next 4 columns present the numbers of vulnerabilities detected by each engine according to vulnerability category: null dereferences (ND), uninitialized variables (UV), dead stores (DS), resource/memory leaks (RL/ML). For each category, we present the number of vulnerabilities each engine reported, as well as true positives (TPs) among them. The last 5 columns present the total vulnerabilities detected with true positives, as well as the metrics of precision, recall and F1.

As prior studies [37] demonstrated, static analysis may have high false positives. Hence, to calculate precision, we formed a group with three graduate students who are familiar with C/C++ languages and vulnerability analysis tools, to conduct a manual inspection of the original vulnerability report for each engine, according to the ISO/IEC 9899:2018 standard. After three weeks, they identified all true positives (TPs) as presented in TABLE III.

Empirical results give interesting findings and insights. First, except for Infer, Clang and CppCheck, other security analysis engines have a low precision (under 30%). Although CppCheck has the highest precision (45.28%), its recall is rather low (4.12%), thus it is F1 score is only 7.56%. Both Infer and Clang have not only high recalls (41.07% and 46.74%, respectively), but also high F1 scores (38.09% and 44.05%, respectively), which results in their high accuracy.

Second, to further explore false positives and false negatives, the aforementioned group of graduate students conducted a

TABLE IV
PYTHON/C API-RELATED VULNERABILITIES DETECTED BY PVMSCAN

Tools	ND		UV		DS		RL/ML		Total				
	Report	TPs	Report	TPs	Report	TPs	Report	TPs	Report	TPs	Precision	Recall	F1
Infer	129	47	34	19	4	4	0	0	167	70	41.92%	12.03%	18.69%
Clang	6	4	0	0	2	2	0	0	8	6	75.00%	1.03%	2.03%
TscanCode	20	8	3	1	0	0	0	0	23	9	39.13%	1.55%	2.98%
Fortify	8	4	0	0	0	0	4	0	12	4	33.33%	0.69%	1.35%
CppCheck	5	2	5	2	0	0	0	0	10	4	40.00%	0.69%	1.35%
Coverity	3	3	0	0	0	0	0	0	3	3	100.00%	0.52%	1.03%
Valgrind	0	0	0	0	0	0	0	0	0	0	\	0.00%	\
FlawFinder	0	0	0	0	0	0	0	0	0	0	\	0.00%	\
Total	171	68	42	22	6	6	4	0	223	96	\	\	\

manual analysis of the corresponding code fragments (in another two weeks). This inspection reveals two key reasons for false positives: 1) duplicated reports: for example, we found 349 out of the 608 (57.40%) dead store vulnerabilities reported by Clang were duplicates; and 2) missing context information: for example, the Infer engine reported a null dereference vulnerability for the variable pointer `item` at line 6 in the following code snippet. However, in fact, the `assert(item != NULL)` at line 5 guarantees the `item` cannot be `NULL`. Fortunately, most of analysis engines are

```
1 //Python-3.10\Modules\_collectionsmodule.c
2 static int deque_del_item(...){
3   PyObject *item;
4   ...
5   assert (item != NULL);
6   Py_DECREF(item);
7   ... }
```

still under active development and future improvements may reduce such false positives. On the other hand, we observed that all 8 engines reported false negatives. For example, there is a known Python issue (bpo-41175 [105]) as demonstrated by the following code snippet, where the pointer `result->ob_bytes` is nullable. Unfortunately, none of the

```
1 //Python-3.10\Objects\bytearrayobject.c
2 PyObject *PyByteArray_Concat(...){...
3 result = ...;
4 if (result != NULL) {
5  memcpy(result->ob_bytes,va.buf,va.len);
6  ...}...}
```

8 engines detected it. We argue that one key reason for such false negatives is that these engines do not cover all possible execution paths due to the nature of static analysis, hence path-dependent vulnerabilities are hard to detect under some complex situations.

Third, none of these engines detected true positives for memory/resource leaks. We believe the reason is that CPython makes use of a well-designed memory management strategy based on arenas, pools, and blocks [106]. All allocation functions are classified into 3 categories: Raw, Mem, and Object, which represent different allocation strategies. Moreover, CPython makes use of reference counting and garbage collections to recycle useless memory [107], thus memory/resource leaks unlikely happen in practice.

Finally, Both Valgrind and FlawFinder detected no true positives. For Valgrind, the Python development team acknowledged us that they already deployed Valgrind. And for FlawFinder, its manual specifies that it makes use of a simple pattern-matching strategy which may be too weak for large code bases like PVMs.

Summary: the official CPython PVM is vulnerable, with 582 vulnerabilities detected. And both Infer and Clang are superior to other analysis engines with higher F1 scores.

F. *Python/C API-related vulnerabilities*

To answers **RQ2** by investigating the effectiveness of PVMSCAN to detect Python/C API-related vulnerabilities, we present, in TABLE IV, the results of applying PVMSCAN to scan Python/C APIs in the latest CPython version 3.10.

These empirical results give interesting findings and insights. First, 96 out of 582 (16.49%) vulnerabilities are Python/C API-related. Furthermore, among all 96 vulnerabilities, 68 (70.83%) are null dereferences. A careful source code inspection reveals the root cause: many Python/C APIs, such as `Py_INCREF()` and `Py_DECREF()`, take pointers as arguments and require that the incoming pointers to be non-nullable. Unfortunately, for many use cases, CPython developers overlook these security requirements, and the lack of security checking leads to null dereference vulnerabilities.

Second, to evaluate the effectiveness of PVMSCAN to process the Python/C API-related vulnerabilities, our group manually compared the original vulnerability report (generated by the vulnerability analysis module) and the vulnerability summary (generated by the purification module). Our analysis demonstrated that the purification module identified 127

TABLE V
EVOLUTION OF VULNERABILITIES, IN 11 VERSIONS OF CPYTHON

Versions	ND		UV		DS		RL/ML		Total		Files	LoC	VPTL	
	TPs	Py/C	TPs	Py/C	TPs	Py/C	TPs	Py/C	TPs	Py/C			TPs	Py/C
3.0	19	16	27	4	47	2	1	1	94	23	315	248,203	0.38	0.09
3.1	21	18	31	4	50	2	0	0	102	24	318	256,181	0.40	0.09
3.2	14	11	36	6	20	1	0	0	70	18	329	275,284	0.25	0.07
3.3	13	11	57	11	8	3	1	0	79	25	335	313,723	0.25	0.08
3.4	17	14	64	18	29	2	2	0	112	34	341	329,420	0.34	0.10
3.5	22	18	55	15	27	2	1	0	105	35	351	352,786	0.30	0.10
3.6	28	24	60	22	37	3	1	0	126	49	356	364,868	0.35	0.13
3.7	31	21	60	20	38	3	0	0	129	44	362	387,620	0.33	0.11
3.8	30	24	64	16	111	3	0	0	205	43	358	407,756	0.50	0.11
3.9	35	27	73	19	111	4	0	0	219	50	361	423,721	0.52	0.12
3.10	55	47	86	19	98	4	0	0	239	70	344	479,044	0.50	0.15
Total	285	231	613	154	576	29	6	1	1480	415	3770	3,838,606	0.39	0.11

(56.95%) false positives in the original vulnerability report. This finding gives evidence for the importance of purification: without this phase, these Python/C API-related false positives would be (incorrectly) fixed thus leading to redundant patches.

Summary: Python/C API-related vulnerabilities constitute a large proportion (16.49%) of all vulnerabilities, and most of them (70.83%) are null dereference vulnerabilities.

G. Evolution

To answer **RQ3** by investigating the evolution of vulnerabilities in different versions of Python virtual machines, we applied PVMSCAN to 11 versions of CPython (from 3.0 to the latest 3.10). Due to space limit, we only present numbers of true positives (TPs), as well as numbers of Python/C API-related ones (Py/C) from Infer. The reproduction package contains all original data.

Several interesting findings and insights can be obtained from these empirical results. First, in order to gain an understanding of how the overall code quality of CPython VM evolves, we calculate the numbers of vulnerabilities per thousand line (VPTL), which is a well-established metrics for measuring code quality [108] [109] [110]. Although the CPython has grown significantly, from 248,203 lines of code in 3.0 to 479,044 lines of code in 3.10, the VPTL has grown slowly from 0.38 to 0.50 (that is, about 1 vulnerability in 2000 lines of code). This finding indicates that the overall quality of CPython remained relatively stable.

Second, in all 11 versions from 3.0 to 3.10, Python/C API-related null dereference (ND) vulnerabilities (231) constitute a major portion of all 285 NDs (83.15%), which is in par with the finding in RQ2. A further source code inspection reveals the following key reason: Python/C APIs have distinct security requirements for function arguments or return values. Unfortunately, these requirements have not been well satisfied in practice [62], which lead to null dereference vulnerabilities.

Summary: the code quality of CPython VM remains stable overall, even though its code grows significantly from 248,203 to 479,044 LOC; and Python/C API-related vulnerabilities (415) constitute a large portion of all 1480 vulnerabilities (28.04%).

H. Automatic Rectification

To answers **RQ4** by investigating the effectiveness of PVMSCAN to rectify vulnerabilities automatically, we present in TABLE VI the rectification results for CPython 3.10.

These empirical results give several relevant findings. First, the automatic rectification is quite effective. As the 3rd column of TABLE VI shows, PVMSCAN successfully rectified 166 out of 239 (69.46%) vulnerabilities, including 54 null dereferences (98.18%), 85 uninitialized variables (98.84%) and 27 dead stores (27.55%). For Python/C API-related vulnerabilities, PVMSCAN successfully rectified 66 out of 70 (94.29%) vulnerabilities.

TABLE VI
VULNERABILITIES PVMSCAN RECTIFIED IN CPYTHON 3.10

Vulnerability Category	No.	Rectified	Python/C API-related	Rectified
ND	55	54 (98.18%)	47	47 (100%)
UV	86	85 (98.84%)	19	19 (100%)
DS	98	27 (27.55%)	4	0 (0%)
Total	239	166 (69.46%)	70	66 (94.29%)

Second, in order to gain an understanding why PVMSCAN failed to rectify some vulnerabilities (especially the DS category), our group further conducted a manual inspection of all the vulnerable code segments that PVMSCAN failed to rectify. This inspection reveals a key reason: as prior studies [41] [43] has demonstrated, aggressive dead store elimination may remove seemingly useless but actually useful memory stores. For instance, in multi-threaded programming, developers often add dead stores into specific threads to reduce thread contention. Hence, to avoid removing useful dead stores, PVMSCAN employed a heuristics to perform conservative removal, which leads to a low DS rectification ratio.

Finally, to testify the performance of automatic rectification, we conducted experiments to measure the time PvmScan spent in rectifying 11 different versions (from 3.0 to 3.10). We ran PvmScan 30 rounds on each version of CPython, respectively. Then, we calculated the time spent for each run, and the time spent to rectify each vulnerability. TABLE VII presents the number of rectified vulnerabilities (#Rectified), the total time to rectify all vulnerabilities, and the average time to rectify one vulnerability, all in milliseconds. PvmScan is efficient: it takes only 416.52 milliseconds to rectify all 166 vulnerabilities in version 3.10. In the meanwhile, it takes about 2 to 3 milliseconds to rectify one vulnerability.

TABLE VII
Performance of PvmScan on 11 CPython VMs

Versions	#Rectified	Total Time (ms)	Average Time (ms)
3.0	62	161.00	2.60
3.1	66	166.29	2.52
3.2	56	191.43	3.42
3.3	58	193.32	3.33
3.4	83	285.42	3.44
3.5	56	274.74	3.19
3.6	94	320.71	3.41
3.7	107	367.58	3.44
3.8	110	411.79	3.74
3.9	123	396.30	3.22
3.10	166	416.52	2.51

Summary: most vulnerabilities in CPython are straightforward to rectify: PvmScan successfully rectified 69.46% vulnerabilities, and 94.29% Python/C API-related ones. PvmScan is efficient and practical to rectify vulnerabilities.

I. *Usefulness*

To answer **RQ5** by investigating the usefulness of PvmScan, we sent part of our initial empirical study results to the CPython development team via a Python issue 46280[1].

The developers of CPython carefully reviewed the Python issue we reported, acknowledged us, confirmed and already fixed 2 bugs based on our issue report while others are still being analyzed. For fairness, we sent the original vulnerability report without filtering false positives manually.

```
1 //Python-3.10\Modules\_tracemalloc.c#line1242
2 ... *traces2=tracemalloc_copy_traces(traces);
3 ++ if (traces2 == NULL) {
4 ++ return -1;
5 ++ }
6    if (...){
7     _Py_hashtable_destroy(traces2);
8     return -1;}
```

Interestingly, the way CPython developers fix the reported bugs is very similar to the one proposed by PvmScan's automatic rectification, as their bp-46280 commits[2] [3] demonstrated for the above code snippet.

[1]https://bugs.python.org/issue46280

In the meanwhile, the CPython development team has similar findings as in this work: 1) there are false positives in the original Python issue report (nevertheless, they do not give quantitative analysis as we did in this work); and 2) for some vulnerabilities such as dead stores, CPython development team admitted that "we've discussed this before. The consensus last time was to leave code like this in place." CPython developers are also conservative about removing the dead store vulnerabilities.

Summary: CPython development team acknowledged our issue reports, and confirmed and fixed 2 bugs by the essentially same techniques as our automatic rectifications.

V. Implications

This section discusses some implications of this work, along with some important directions for future research.

For PVM Maintainers. Results in this work provide PVM maintainers important insights into improving the security of the large PVM code bases. On one hand, security engines already deployed have improved the security of PVM considerably. For example, the Valgrind engine does not report any vulnerabilities, because the CPython development team acknowledged us that they already make use of Valgrind internally to scan the source code during development. On the other hand, the latest progress in security analysis techniques and tools will benefit the PVM maintainers. For example, based on our reports and findings, the PVM maintainers are planning to leverage latest security tools (e.g., Infer) in their development. Another direction for exploration is to integrate tools like PvmScan into the production systems (e.g., CI/DI), which can further improve the security of PVMs.

For Security Analysis Tool Builders. Given the importance of software infrastructures like PVMs, results in this work provide insights to tool builders to develop more effective security tools. On one hand, security tool builders should put more research efforts into detection algorithms for Python/C APIs. Given the language dependent nature of Python/C APIs, one plausible solution is to introduce security tool plugins (e.g., IDAPro plugins [44]). On the other hand, security tool builders should further investigate techniques to scan multi-lingual applications, by inter-language program analysis [48].

VI. Threats to Validity

As in any empirical study, there are threats to validity with our work. We attempt to remove these threats where possible, and mitigate the effect when removal is not possible.

Data sets. In this work, we have used CPython as our data set to evaluate PvmScan, as it is the official and most deployed PVM. In the meanwhile, there are other Python virtual machines such as PyPy [51], Graalpython [96], IronPython [76]. Fortunately, the modular design of PvmScan make it

[2]https://github.com/python/cpython/pull/30592
[3]https://github.com/python/cpython/pull/30593

straightforward to study other PVMs, and we have made our tool open source and publicly available. In the short term, we are planning to conduct experiments on MicroPython [84], a widely used Python VM targeting microcontrollers.

Other Vulnerabilities. In this work, we have concentrated on detecting and rectifying memory vulnerabilities in Python virtual machines, and the empirical results demonstrated that PVMSCAN is effective in achieving this research goal. Although memory vulnerabilities are among one of the most severe vulnerabilities [68], there are other types of vulnerabilities in Python virtual machines such as buffer overflows, thread safety, information flow vulnerabilities. Fortunately, the architecture of PVMSCAN (Fig. 1) can be easily extended to support the detection and rectification of other types of vulnerabilities without difficulty. For example, to detect and rectify buffer overflows, it is only necessary to extend the rectification module to insert range checking code for buffer accesses, with other modules of PVMSCAN remained unchanged, as long as the analysis engines are able to detect buffer overflows.

Analysis Engines. In this work, we used 8 analysis engines to obtain the empirical results. Although the 8 analysis engines we used in our work are widely used thus empirical results are trustworthy, there are proprietary analysis engines such as CxSAST [80] which may detect other vulnerabilities, but we do not have the necessary resources to explore proprietary engines. Fortunately, the architecture of PVMSCAN (Fig. 1) is neutral to the specific analysis engines used. And the modular design of PVMSCAN makes it easy to incorporate other analysis engines.

VII. RELATED WORK

In recent years, there are a significant amount of studies on security of virtual machines. However, the work in this paper stands for a novel contribution to this field.

Native Code Security. There has been many research on native code security. CCured [81] is a program transformation system providing security guarantees for C programs. Cyclone [82] is a safe dialect of C to prevent the C programs from vulnerabilities such as buffer overflows while retaining the syntax and semantics of C language. Frank and Red [83] presented Mudflap, to transparently adds protective code to potentially unsafe C/C++ programs. Gregory and Roland [85] presented EffectiveSan, an technology for dynamically typed C/C++ called to ensure type and memory safety. Xu and Ren et al. [86] proposed a dynamic memory error detection method based on dynamic binary translation to analyze heap and stack memory destruction. Moritz et al. [87] presented HEAPHOPPER to automatically analyze the exploitability of heap implementations and find weaknesses. A major limitation of these studies is that they only consider the vulnerabilities in general native code, but did not study the vulnerabilities in the CPython virtual machine and did not address the challenges posed by Python/C APIs.

Foreign Function Interface Security. There has been a lot of work on Foreign Function Interface (FFI) security. Furr et al. [55] [56] presented a type inference system to check the OCaml/C interface. They later extended it to check the type safety of programs that use JNIs. Tan et al. [57] proposed a framework called SafeJNI that ensures type safety of heterogeneous programs containing Java and C components. Li et al. [58] [59] [60] [61] proposed a static analysis framework to examine exception errors in JNI programs.

Cpychecker, proposed by D. Malcom [63], and Pungi, proposed by S. Li et al. [69], statically detect the reference counting errors in Python/C programs. Mao et al. [70] proposed RID, an inconsistent path pair checking technology to statically discover the reference vulnerabilities. Hu et al. [62] implemented a tool PyCEAC to study the evolution of Python/C APIs, and proposed 10 classes of vulnerability patterns. Jiang et al. [27] proposed a framework called PyGuard to find and understand real-world security vulnerabilities in the CPython virtual machines. However, a major limitation of existing work is that they can only detect vulnerabilities, but did not study the problem of automatic rectifications.

Automatic Program Rectification. Recently, automatic program rectification has become one of the most important subjects. Qing et al. [89] proposed a program repair system called ACS, to generate precise conditions at faulty locations. Tonder et al. [90] presented an automatic program repair technique using separate logic to find and fix vulnerabilities related to general pointer safety properties without test cases. Gupta et al. [91] presented DeepFix to fix errors in C program based on a multi-layered sequence-to-sequence neural network. Yan et al. [92] proposed AutoFix to fix memory leaks for C programs by combining static and dynamic program analysis. Cheng et al. [93] proposed CIntFix to automatically fix integer errors for C by replacing original C integers with dynamic-precision integers. Ke et al. [94] proposed SearchRepair that uses a large body of existing open-source code to find potential fixes. However, existing work only focused on the automatic rectification of vulnerabilities in general small programs. They cannot be used directly to rectify Python/C API related vulnerabilities.

VIII. CONCLUSION

Python will continue to be a dominant language in the era of data science, thus the underlying PVMs should be trustworthy and reliable. In this work, we present the first empirical study of Python virtual machine security. By utilizing a novel tool PVMSCAN, we performed a comprehensive study of the CPython. The empirical results show that the CPython is still vulnerable, and Python/C API-related vulnerabilities constitute a large proportion. Most vulnerabilities in CPython can be easily rectified by PVMSCAN. Our results will benefit PVM maintainers, Python developers, and security tool builders.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their valuable feedback. This work is supported by a graduate education innovation program of University of Science and Technology of China under grant No. 2020YCJC41, No. 2021YCJC34.

REFERENCES

[1] Top Programming Languages 2021:Python dominates as the de facto platform for new technologies. https://spectrum.ieee.org/top-programming-languages-2021
[2] October Headline: Python programming language number 1. https://www.tiobe.com/tiobe-index/
[3] 2021 Developer Survey. https://insights.stackoverflow.com/survey/2021
[4] Manipulating machine learning: Poisoning attacks and countermeasures for regression learning. 2018 IEEE Symposium on Security and Privacy (SP), 2018: 19-35.
[5] Xiao H, Biggio B, Brown G, et al. Is feature selection secure against training data poisoning?. International Conference on Machine Learning, 2015: 1689-1698.
[6] Mei S, Zhu X. Using machine teaching to identify optimal training-set attacks on machine learners. Twenty-Ninth AAAI Conference on Artificial Intelligence, 2015.
[7] Duncan J, Kapoor R, Agarwal A, et al. VeridicalFlow: a Python package for building trustworthy data science pipelines with PCS. Journal of Open Source Software, 2022, 7(69): 3895.
[8] Calix R A, Singh S B, Chen T, et al. Cyber security tool kit (CyberSecTK): A Python library for machine learning and cyber security. Information, 2020, 11(2): 100.
[9] Python Developers Survey 2020. https://www.jetbrains.com/zh-cn/lp/python-developers-survey-2020/
[10] Python/C API Reference Manual. https://docs.python.org/3/c-api/index.html
[11] Nagpal A, Gabrani G. Python for data analytics, scientific and technical applications. 2019 Amity international conference on artificial intelligence (AICAI). IEEE, 2019: 140-145.
[12] McKinney W. pandas: a foundational Python library for data analysis and statistics. Python for high performance and scientific computing, 2011, 14(9): 1-9.
[13] Aslam F A, Mohammed H N, Mohd J M, et al. Efficient way of web development using python and flask. International Journal of Advanced Research in Computer Science, 2015, 6(2): 54-57.
[14] Grinberg M. Flask web development: developing web applications with python. " O'Reilly Media, Inc.", 2018.
[15] Raschka S, Mirjalili V. Python Machine Learning: Machine Learning and Deep Learning with Python. Scikit-Learn, and TensorFlow. Second edition ed, 2017.
[16] Raschka S, Patterson J, Nolet C. Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. Information, 2020, 11(4): 193.
[17] Aggarwal A, Garhwal S, Kumar A. HEDEA: a Python tool for extracting and analysing semi-structured information from medical records. Healthcare informatics research, 2018, 24(2): 148-153.
[18] Widodo C E, Adi K, Gernowo R. Medical image processing using python and open cv. Journal of Physics: Conference Series. IOP Publishing, 2020, 1524(1): 012003.
[19] Aziz M V G, Prihatmanto A S, Hindersah H. Implementation of lane detection algorithm for self-driving car on toll road cipularang using Python language. 2017 4th international conference on electric vehicular technology (ICEVT). IEEE, 2017: 144-148.
[20] Chishti S O A, Riaz S, BilalZaib M, et al. Self-driving cars using CNN and Q-learning. 2018 IEEE 21st International Multi-Topic Conference (INMIC). IEEE, 2018: 1-7.
[21] CPython. https://github.com/python/cpython
[22] CVE-2021-23336. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-23336.
[23] CVE-2018-1000117. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1000117.
[24] CVE-2017-1000158. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1000158.
[25] CVE-2016-5636. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5636.
[26] CVE-2021-3177. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3177
[27] Chengman J, Baojian H, Wanrong O, et al. PyGuard: Finding and Understanding Vulnerabilities in Python Virtual Machines. International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2021
[28] Cannon B, Wohlstadter E. Controlling access to resources within the python interpreter. Proceedings of the Second EECE, 2010, 512: 1-8.
[29] Berdine J, Calcagno C, O'Hearn P W. Smallfoot: Modular automatic assertion checking with separation logic. International Symposium on Formal Methods for Components and Objects. Springer, Berlin, Heidelberg, 2005: 115-137.
[30] Tencent. TscanCode. https://github.com/Tencent/TscanCode.
[31] Clang Static Analyzer. https://clang-analyzer.llvm.org/
[32] Coverity. https://scan.coverity.com/.
[33] Fortify. https://www.joinfortify.com/.
[34] Valgrind. https://valgrind.org/.
[35] Wheeler, D. A. Flawfinder. http://www.dwheeler.com/flawfinder/.
[36] Daniel Marjamki. Cppcheck. http://cppcheck.sourceforge.net/.
[37] Tan, G. and Croft, J. (2008). An empirical security study of the native code in the JDK. In 17th Usenix Security Symposium, pages 365–377.
[38] Python/C API Reference Manual. https://docs.python.org/3/c-api/index.html
[39] CWE-476: NULL Pointer Dereference. https://cwe.mitre.org/data/definitions/476.html
[40] CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html
[41] D'Silva V, Payer M, Song D. The correctness-security gap in compiler optimization. 2015 IEEE Security and Privacy Workshops. IEEE, 2015: 73-87.
[42] Gabel M, Yang J, Yu Y, et al. Scalable and systematic detection of buggy inconsistencies in source code. Proceedings of the ACM international conference on Object oriented programming systems languages and applications. 2010: 175-190.
[43] Yang Z, Johannesmeyer B, Olesen A T, et al. Dead store elimination (still) considered harmful. 26th USENIX Security Symposium (USENIX Security 17). 2017: 1025-1040.
[44] IDAPro Plugings. https://github.com/onethawt/idaplugins-list
[45] CWE-457: Use of Uninitialized Variable. https://cwe.mitre.org/data/definitions/457.html
[46] Cho H, Park J, Kang J, et al. Exploiting Uses of Uninitialized Stack Variables in Linux Kernels to Leak Kernel Pointers. 14th USENIX Workshop on Offensive Technologies (WOOT 20). 2020.
[47] Flake H. Attacks on uninitialized local variables. Black Hat Europe, 2006.
[48] Gang Tan and Greg Morrisett. 2007. Ilea: inter-language analysis across java and c. In Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications (OOPSLA '07). Association for Computing Machinery, New York, NY, USA, 39–56. DOI:https://doi.org/10.1145/1297027.1297031
[49] CWE-401: Missing Release of Memory after Effective Lifetime https://cwe.mitre.org/data/definitions/401.html
[50] CWE-772: Missing Release of Resource after Effective Lifetime https://cwe.mitre.org/data/definitions/772.html
[51] The Pypy VM. https://www.pypy.org/
[52] Jung C, Lee S, Raman E, et al. Automated memory leak detection for production use. Proceedings of the 36th International Conference on Software Engineering. 2014: 825-836.
[53] Milburn A, Bos H, Giuffrida C. Safelnit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities. NDSS. 2017, 17: 1-15.
[54] Stepanov E, Serebryany K. MemorySanitizer: fast detector of uninitialized memory use in C++. 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 2015: 46-55.
[55] Furr M, Foster J S. Checking type safety of foreign function calls. ACM SIGPLAN Notices, 2005, 40(6): 62-72.
[56] Furr M, Foster J S. Polymorphic type inference for the JNI. European Symposium on Programming. Springer, Berlin, Heidelberg, 2006: 309-324.
[57] Tan G, Appel A W, Chakradhar S, et al. Safe Java native interface. Proceedings of IEEE International Symposium on Secure Software Engineering. 2006, 97: 106.
[58] Li S, Tan G. Finding bugs in exceptional situations of JNI programs. Proceedings of the 16th ACM conference on Computer and communications security. 2009: 442-452.
[59] Li S, Tan G. JET: exception checking in the java native interface. ACM SIGPLAN Notices, 2011, 46(10): 345-358.
[60] Li S, Tan G. Exception analysis in the java native interface. Science of Computer Programming, 2014, 89: 273-297.
[61] Tan G, Croft J. An Empirical Security Study of the Native Code in the JDK. Usenix Security Symposium. 2008: 365-378.

[62] Hu M, Zhang Y. The Python/C API: Evolution, Usage Statistics, and Bug Patterns. 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2020: 532-536.
[63] D. Malcom. a static analysis tool for cpython extension code. https://gcc-python-plugin.readthedocs.org/en/latest/cpychecker.html.
[64] Foster J C, Osipov V, Bhalla N, et al. Buffer overflow attacks. Syngress, Rockland, USA, 2005.
[65] Dietz W, Li P, Regehr J, et al. Understanding integer overflow in C/C++. ACM Transactions on Software Engineering and Methodology (TOSEM), 2015, 25(1): 1-29.
[66] The Python bytecode. https://docs.python.org/3/library/dis.html#python-bytecode-instructions
[67] Zhang H, Wang S, Li H, et al. A Study of C/C++ Code Weaknesses on Stack Overflow. IEEE Transactions on Software Engineering, 2021.
[68] Younan Y, Joosen W, Piessens F, et al. Security of memory allocators for C and C++. Technical Report CW 419, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, 2005.
[69] Li S, Tan G. Finding reference-counting errors in Python/C programs with affine analysis. European Conference on Object-Oriented Programming. Springer, Berlin, Heidelberg, 2014: 80-104.
[70] Mao J, Chen Y, Xiao Q, et al. RID: finding reference count bugs with inconsistent path pair checking. Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems. 2016: 531-544.
[71] Simons A J H. Borrow, copy or steal? loans and larceny in the orthodox canonical form. Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. 1998: 65-83.
[72] Wei Y, Pei Y, Furia C A, et al. Automated fixing of programs with contracts. Proceedings of the 19th international symposium on Software testing and analysis. 2010: 61-72.
[73] Arcuri A, Yao X. A novel co-evolutionary approach to automatic software bug fixing. 2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence). IEEE, 2008: 162-168.
[74] Weimer W, Nguyen T V, Le Goues C, et al. Automatically finding patches using genetic programming. 2009 IEEE 31st International Conference on Software Engineering. IEEE, 2009: 364-374.
[75] Qi Y H, Mao X G, Wen Y J, et al. More efficient automatic repair of large-scale programs using weak recompilation. Science China Information Sciences, 2012, 55(12): 2785-2799.
[76] The Ironpython VM. https://ironpython.net/
[77] Xiong Y, Hu Z, Zhao H, et al. Supporting automatic model inconsistency fixing. Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. 2009: 315-324.
[78] Python. https://www.python.org/
[79] Infer. https://fbinfer.com/
[80] CxSAST. https://checkmarx.com/
[81] Necula G C, Condit J, Harren M, et al. CCured: Type-safe retrofitting of legacy software. ACM Transactions on Programming Languages and Systems (TOPLAS), 2005, 27(3): 477-526.
[82] Jim T, Morrisett J G, Grossman D, et al. Cyclone: a safe dialect of C. USENIX Annual Technical Conference, General Track. 2002: 275-288.
[83] Eigler F C. Mudflap: Pointer use checking for c/c+. Proceedings of the First Annual GCC Developers' Summit, 2003: 57-70.
[84] MicroPython. http://micropython.org/
[85] Duck G J, Yap R H C. EffectiveSan: type and memory error detection using dynamically typed C/C++. Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2018: 181-195.
[86] Xu H, Ren W, Liu Z, et al. Memory Error Detection Based on Dynamic Binary Translation. 2020 IEEE 20th International Conference on Communication Technology (ICCT). IEEE, 2020: 1059-1064.
[87] Eckert M, Bianchi A, Wang R, et al. Heaphopper: Bringing bounded model checking to heap implementation security[C]//27th USENIX Security Symposium (USENIX Security 18). 2018: 99-116.
[88] Gao Q, Xiong Y, Mi Y, et al. Safe memory-leak fixing for c programs. 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. IEEE, 2015, 1: 459-470.
[89] Xiong Y, Wang J, Yan R, et al. Precise condition synthesis for program repair. 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE, 2017: 416-426.
[90] van Tonder R, Goues C L. Static automated program repair for heap properties. Proceedings of the 40th International Conference on Software Engineering. 2018: 151-162.
[91] Gupta R, Pal S, Kanade A, et al. Deepfix: Fixing common c language errors by deep learning. Thirty-First AAAI Conference on Artificial Intelligence. 2017.
[92] Yan H, Sui Y, Chen S, et al. Automated memory leak fixing on value-flow slices for c programs. Proceedings of the 31st Annual ACM Symposium on Applied Computing. 2016: 1386-1393.
[93] Cheng X, Zhou M, Song X, et al. Automatic fix for C integer errors by precision improvement. 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC). IEEE, 2016, 1: 2-11.
[94] Ke Y, Stolee K T, Le Goues C, et al. Repairing programs with semantic code search (t). 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2015: 295-306.
[95] Sider. https://sider.review/
[96] The graalpython VM. https://github.com/oracle/graalpython
[97] Cousot P, Cousot R. Modular static program analysis. International Conference on Compiler Construction. Springer, Berlin, Heidelberg, 2002: 159-179.
[98] Python Issue 46280. https://bugs.python.org/issue46280
[99] bpo-46280. https://github.com/python/cpython/commit/86d18019e96167c5ab6f5157fa90598202849904
[100] Ben Asher Y, Rotem N. The effect of unrolling and inlining for Python bytecode optimizations. Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference. 2009: 1-14.
[101] C API Stability. https://docs.python.org/3/c-api/stable.html
[102] Py_CLEAR(). https://docs.python.org/zh-cn/3/c-api/refcounting.html?highlight= py_clear#c.Py_CLEAR
[103] CWE-369: Divide By Zero. https://cwe.mitre.org/data/definitions/369.html
[104] The Log4j vulnerability. CVE-2021-44832. https://logging.apache.org/log4j/2.x/security.html
[105] bpo-41175. https://github.com/python/cpython/pull/21240.
[106] Python Memory Management. https://docs.python.org/3/c-api/memory.html
[107] Perl T. Python Garbage Collector Implementations CPython, PyPy and GaS. 2012.
[108] Misra S C, Bhavsar V C. Relationships between selected software measures and latent bug-density: Guidelines for improving quality. International Conference on Computational Science and Its Applications. Springer, Berlin, Heidelberg, 2003: 724-732.
[109] Bach T, Andrzejak A, Pannemans R, et al. The impact of coverage on bug density in a large industrial software project. 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, 2017: 307-313.
[110] Shams Z, Edwards S H. An experiment to test bug density in students' code. Proceeding of the 44th ACM technical symposium on Computer science education. 2013: 742-742.