PyGuard: Finding and Understanding Vulnerabilities in Python Virtual Machines

Chengman Jiang   Baojian Hua*   Wanrong Ouyang   Qiliang Fan   Zhizhong Pan
School of Software Engineering
University of Science and Technology of China
{sa148, oywr, sa613162, sg513127}@mail.ustc.edu.cn   bjhua@ustc.edu.cn*

***Abstract*—Python has become one of the most popular programming languages in the era of data science and machine learning, and is also widely deployed in safety-critical fields like medical treatment, autonomous driving systems, etc. However, as the official and most widely used Python virtual machine, CPython, is implemented using C language, existing research has shown that the native code in CPython is highly vulnerable, thus defeats Python's guarantee of safety and security. This paper presents the design and implementation of PyGuard, a novel software prototype to find and understand real-world security vulnerabilities in the CPython virtual machines. With PyGuard, we carried out an empirical study of 10 different versions of CPython virtual machines (from version 3.0 to the latest 3.9). By scanning a total of 3,358,391 lines native code, we have identified 598 new vulnerabilities. Based on our study, we describe a taxonomy to classify vulnerabilities in CPython virtual machines. Our taxonomy provides a guidance to construct automated and accurate bug-finding tools. We also suggest systematic remedies that can mediate the threats posed by these vulnerabilities.**

***Index Terms*—Python, Virtual machines, Vulnerabilities, Bug taxonomy, Program analysis**

I. Introduction

Python has become a very popular programming language since its birth in late 80s last century, with wide applications in data science, security analysis, and machine learning. Python uses a combination of dynamic typing and garbage collection to guarantee both runtime type safety and memory safety, which is important in safety-critical applications such as autonomous driving systems. As a result, existing research has proven that Python is safe [1] [2].

Unfortunately, as Python is an interpreted language, and the official Python virtual machine, CPython, is implemented in C and consists of more than 420,000 lines of native code (the latest CPython version 3.9). Existing research has shown that C/C++ code is vulnerable [3] to common program bugs like buffer overflows, double free, dangling pointers, etc, the vulnerabilities of Python virtual machines will defeat Python's guarantees of safety and security [4], [5], [6], [7].

Fig. 1 presents an instance of insufficient error check in CPython version 3.7. The Python/C native interface method `Py_BuildValue` returns a Python object `reduce_value`. If the `Py_BuildValue` function throws an exception, the

```
1.// cpython-3.7\Modules\_pickle.c
2.save_global(PicklerObject *self, PyObject *obj, PyObject *name){
3.    if (parent != module) {
4.        PickleState *st = _Pickle_GetGlobalState();
5.        PyObject *reduce_value = Py_BuildValue("(O(OO))",
6.                                          st->getattr, parent, lastname);
7.++    if (reduce_value == NULL)
8.++        goto error;
9.    status = save_reduce(self, reduce_value, NULL);
10.    ...
11.error:
12.    status = -1;
13.    return status;
14.}
```

Fig. 1: A bug of insufficient error check from CPython version 3.7.

variable `reduce_value` is assigned the value `NULL`. Therefore, without the explicit security patch at lines 7 and 8, the function call at line 9 is vulnerable.

Based on such observations, one natural question to ask is why does the CPython virtual machine still contain such kind of vulnerabilities, given the fact that this virtual machine is relatively mature and is believed to have been thoroughly tested? We believe there are two key technical challenges: first, it's challenging to guarantee the security for native code, due to the inherent vulnerabilities of C/C++, such as integer overflow, formatted string attacks, buffer overflows, heap overflows, race conditions, etc, which are difficult to find and repair. Although there has been a great deal of studies for these vulnerabilities in general C/C++ code, there has been no systematic study on such vulnerabilities in Python virtual machine native code. Furthermore, there has no research on the the classification of bug patterns in native code of Python virtual machines.

Second, Python code may interact with native code through the Python/C native method interfaces. Due to the intrinsic differences between managed Python code and native C code in term of memory model, data representation, semantics, etc, vulnerabilities may be introduced by such interactions. There has been little research on finding and understanding such vulnerabilities in these interactions.

To address these technical challenges, this paper presents a novel security framework **PyGuard** to scan Python virtual machine native code. In PyGuard, we use a combination of static program analysis, differential testing, and manual code inspection technology, to carry out an analysis of the

* Corresponding author.

native code and interface code in Python virtual machines. For native code, we leverage the existing program analysis techniques and tools. For the Python native method interface code, we designed an analysis algorithm, and implemented it in PyGuard.

In order to illustrate the effectiveness of this infrastructure, we design and implement a software prototype, and use this prototype to conduct experiments on 10 different versions of CPython virtual machines (from version 3.0 to 3.9). Experimental results show that PyGuard is effective in finding vulnerabilities in CPython virtual machines, by scanning a total of 3,358,391 lines of native code, PyGuard found 598 new vulnerabilities.

We systematically studied and analyzed all the vulnerabilities reported by PyGuard, and propose a taxonomy to classify these vulnerabilities. Furthermore, we present suggestions to repair these vulnerabilities. We believe this taxonomy and these suggestions will benefit future studies in this direction.

In this work, we mainly investigate the following three research questions (RQs):

- **RQ1:** What are the vulnerability patterns in the CPython virtual machines?
- **RQ2:** How do vulnerabilities evolve, among different versions of CPython virtual machines?
- **RQ3:** What are the most common vulnerabilities in CPython virtual machines?

To our knowledge, this work is the first empirical security study of the CPython virtual machines from version 3.0 to 3.9. To summarize, the technical highlights and contributions of this paper are:

- we constructed a new software prototype PyGuard, to scan and find vulnerabilities in the CPython virtual machines;
- we conducted a thorough empirical security study of the official CPython virtual machines, by scanning a total of 3,358,391 lines of native code, we found 598 new vulnerabilities;
- we gave a taxonomy to classify the vulnerabilities in CPython virtual machines, and suggest systematic remedies that can mediate the threats of these vulnerabilities.

The rest of this paper is organized as follows. Section II introduces the background and related work on security research of native code in Python virtual machines. Section III presents the design and architecture of PyGuard. Section IV presents experiments and results. Section V discusses some limitations of this work by presenting future research directions, and Section VI concludes.

II. Related Work

There have been a significant of studies on the security of both native code and the native method interfaces of virtual machines. This section introduces the related research.

A. Native code security

There has been a lot of research on native code security. Both CCured [8] and Cyclone [9] present security guarantees for C code, through a combination of static and dynamic checks, while retaining the syntax and semantics of C language. These system can eliminate buffer overflows, format string attacks and many memory management errors in C programs.

Buffer overflow has been well studied. Wang et al. [10] proposed a polymorphic SSP (P-SSP) technology. The key insight is to re-randomize canaries for a new process, to ensure that the attackers' knowledge of the canaries will not accumulate through different experiments. Jang et al. [11] propose a technology of code replacement. Ren [12], William [13] and others propose machine learning and neural network models to detect buffer overflow vulnerabilities.

Memory errors in the C language have been well studied. Gregory and Roland [14] introduced dynamically typed C/C++ to detect memory errors, by dynamically checking the effective type of each object at runtime. Dynamic memory error detection technology is also widely used to locate and repair the code that causes memory errors. Xu et al. [15] proposed a dynamic memory error detection method based on dynamic binary conversion, combining call stack tracing and IR language-level memory error detection. Li et al. [16] proposed Memory Access Integrity (MAI), a dynamic method to detect fine-grained memory access errors in binary executable files.

However, a major limitation of these work is that they only studied vulnerabilities in common native code, but does not discuss the vulnerabilities and patterns in a virtual machine setting. Furthermore, these work don't discuss the taxonomy of vulnerabilities.

B. Virtual machine native method interface security

There has been a lot of research on virtual machines' native method interfaces.

Furr and Foster [17] proposed a multi-language type inference system to check OCaml external function interface calls, in order to prevent memory safety violations. They further extended the system to check the type safety of Java Native Interface (JNI) programs [18].

Tan and Li et al. [19] proposed the SafeJNI framework to guarantee type safety when calling native C methods, by performing static and dynamic checks on C code. Tan and Li [20] [21] [22] built a new static analysis framework for the difference between Java and native method exception handling methods to find exception handling errors in Java native code. And on the basis of these work, further conducted empirical security research [23] on native code in JDK, and put forward some vulnerability modes.

The Python/C API security-related research is rare. Pungi [24] uses affine abstraction to statically analyze the SSA form of the program. RID [25] uses inconsistent path pair checking to relax Pungi's assumptions and improve the accuracy of the analysis. Zhang and Hu [26] used static analysis methods to reveal the evolution of Python/C API, and implemented a tool chain PyCEAC to obtain usage data of Python/C API for seven software systems in different fields.

However, a major drawback of existing work is that they did not study the vulnerabilities in the Python/C interfaces in Python virtual machines.

III. Architecture

This section presents the design principles, architecture and components of the PyGuard software prototype.

We have two important principles guiding the construction of the PyGuard software prototype. First, the architecture of PyGuard must support the analysis of different versions of Python virtual machines. The key technical challenge is that existing CPython virtual machines have very different data structures, code layout, and implementation details, etc, which demands different analysis strategies. Thus, by employing a version-neutral architecture, we are able to perform security analysis and result comparison for different versions of CPython VMs simultaneously. Such kind of comparisons give us more insights about the distributions and evolvements of vulnerabilities.

Second, the PyGuard architecture should be modular, to support the analysis of different security features. There are two key observations for such a design decision. First, many security vulnerabilities are orthogonal, so it's possible to scan such vulnerabilities in a modular way in the first place. Second, the analysis techniques for many security vulnerabilities are continually evolving, thus the modular design can make it much easier to strengthen or expand the software prototype.

Based on the above design goals, we present the PyGuard architecture in Fig. 2. SafePy's architecture can be divided into three major components, the first one is the input processing component, which takes the CPython source code as input and outputs the target CPython native code, according to the configuration file. The second one is the security plug-in module, consisting of core analysis engines. This module leverages existing static analysis tools and also integrates the scanning plug-ins we built, to scan the input source native code and analyze vulnerabilities. The third component is the vulnerability analysis and statistics module, in which arbitration and manual analysis are used to generate a final vulnerability report.

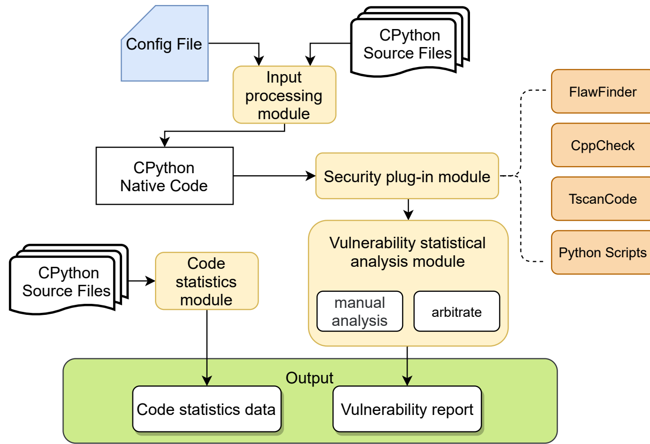

Fig. 2: PyGuard architecture.

In the following several sections, we present the design and implementation details for each component respectively.

A. Input processing

The input processing module will extract both the native code and the native method interface code, from a given Python virtual machine source code. The information that guiding the filtering of the Python VM's target directory is given by the configuration file.

From the architecture perspective, separating this module from other modules has at least two key advantages: first, this design effectively hides the differences between different versions of Python VMs. Second, this module outputs a uniform intermediate representation that subsequent modules will handle, which reduces the implementation efforts of these modules considerably.

B. Security plug-ins

The security plug-in module is the core component of PyGuard, which integrates existing off-the-shelf open source static analysis tools and the scanning plug-in built by us. This module accepts as input the native code generated by the input processing module, analyzes the specific security features, and outputs the scanning analysis results to PyGuard's back-end.

PyGuard makes use of different strategies to scan Python virtual machine native code, and Python native method interface code. To scan common vulnerability patterns in native code, such as buffer overflows, race conditions, and memory management errors, we use a combination of existing tools: FlawFinder [28], CppCheck [29], and TscanCode [30]. FlawFinder uses built-in vulnerability databases such as buffer overflows and formatted string vulnerabilities. FlawFinder scans very fast buts has higher false positives. Both CppCheck and TscanCode use vulnerability rules to support the detection of null pointer, array out of bounds, memory leak, operation error and other kinds of vulnerabilities. TscanCode has a higher accuracy rate than CppCheck. These tools have a high false positive rate. Although some static analysis tools, such as Coverity [31], Checkmarx [32], or Fortify [33], have been proved to generate more accurate results, however, these tools are not easy to be integrated into the open-source PyGuard, due to their commercial licenses.

Some bug patterns in the Python/C native code are particular to the Python native interface, thus existing tools cannot be used to scan these patterns. In order to scan the Python/C native interface methods, we built an analysis plug-in, using Python scripts. To scan Python/C API integer overflow vulnerability, we developed an algorithm to scan all the positions where Python parses parameters and constructs values, then checks whether the format characters will trigger overflow or not.

CPython needs to manage Python memory, which may also contain memory related vulnerabilities. CPython provides specific memory allocators, like `PyMem_RawMalloc/`

`PyMem_RawFree`, `PyMem_Malloc/ PyMem_Free` and `PyMem_MALLOC/ PyMem_FREE`. We build scripts using pattern matching to scan all locations where memory is allocated using these memory allocators, to identify memory management defects. Although this technique is not complicated, but we have found it's quite effective in finding such vulnerabilities.

Native code have different exception handling mechanisms, and the exception thrown should be returned immediately. We developed an algorithm to search all the places where an exception is explicitly thrown. This algorithm searched all the places where `Py_Err` is called and checked to determine whether it returned immediately. Some Python/C APIs use return values to carry internal errors, and such return values should be checked before subsequent operations. We studied the Python language specification and enumerated all APIs that need to be checked for return values. The algorithm also traces all the places where these APIs are called, and check whether the API's return values are checked.

C. Vulnerability statistics

The vulnerability statistical analysis module takes as input the vulnerability data generated by the security plug-in module, and outputs vulnerability report. Technically, this module has two implementation challenges: first, since static analysis tools and security plug-ins have false positives, it's challenging to analyze the vulnerability data to obtain the true security vulnerabilities. Second, how to establish the ground truth for vulnerabilities?

In order to address the first challenge, we adopted the method of arbitration and manual analysis. If a vulnerability is reported by multiple security tools, then it's of high possibility it's a true positive. To address the second challenge, we used a differential comparison technique, i.e., we compared the native code from the Python virtual machine being analyzed with the latest version of the same virtual machine. If there is a security patch for the code in the old VM, the true positive can be confirmed. Although in theory, this judging technique may not be complete, because the new VM may not have all security vulnerabilities fixed, but there is no complete algorithm to detect all possible vulnerabilities in a program as this problem is undecidable. On the other hand, we have found that this method is very effective in practice by confirming a lot of vulnerabilities.

IV. Experiments and Results

This section presents the experiments we conducted using the PyGuard prototype, and analyzes the experimental results. The goal of our experiment is to answer two important questions:

- Does the system work? In particular, will the system be able to detect unknown security vulnerabilities in the actual CPython virtual machine?
- What's the accuracy rate of this system, to be specific, what are false alarm rate and omission alarm rate? How many and what security vulnerabilities can be automatically repaired?

A. Experimental Setup

The experiments are conducted on a server with an AMD Ryzen 7 3750H CPU and 4GB of memory running Ubuntu 14.04.4.

B. Data Sets

We selected 10 CPython virtual machines as data sets, from version 3.0 to 3.9. There are two important considerations to choose the CPython virtual machine. First, CPython is the official Python virtual machine, which is the most widely used. Second, according to the latest Python developer report [27], Python version 3.x usage is over 90% and continues to grow, but Python 2.x is no longer maintained and considered to be obsolete.

CPython is built with C language, and its main code structure are shown in the TABLE I (take CPython 3.7 as an example). CPython code is relatively complex, limited to manual inspection of vulnerabilities. we focused our study on the code under the directories: Includes, Modules, Objects, Parser, PC and Python. Includes is a header package for C; Modules is a standard library file written in C; The Objects package contains all of Python's built-in Objects; The PC contains all Windows compilation support files; Parser is the lexing and parsing part of the Python interpreter; Python is the compiler and execution engine part of the interpreter.

TABLE I: CPython 3.7 code structure.

Module	Files of Native Code	Lines of Native Code
Doc	0	0
Grammar	0	0
Include	99	16442
Lib	0	0
Misc	1	187
Modules	108	189728
Objects	42	88515
Parser	19	5259
PC	20	8985
Programs	3	503
Python	70	78001
Tools	0	0
Total	362	387620

In order to gain a deep understanding of the native code evolution in CPython VMs, we conducted an experiment to investigate the files and code sizes in these VMs. In Fig. 3, we present the data for CPython 3.0 to 3.9. We conclude that the CPython VMs have grown significantly in the 10 versions. From version 3.0 to version 3.9, the number of native files has grown from 315 to 361, and lines of native code have grown from 248203 to 423721. There are a total of 3426 native files and 3,358,391 lines of native code, in all 10 VMs.

C. Experimental results

We used PyGuard to scan and analyze the native code in 10 versions of CPython VMs. In this section, we present the experimental results by answering the research questions.

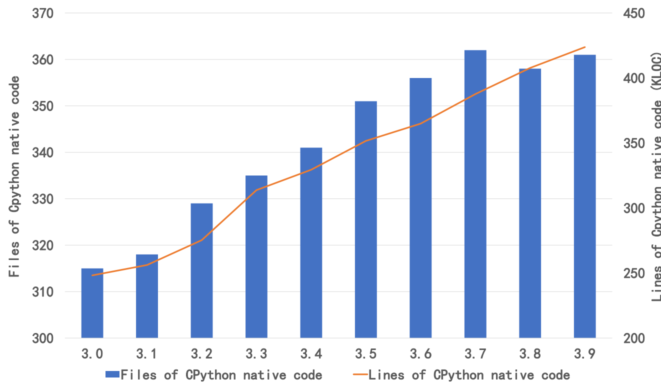

Fig. 3: Files and Code sizes of 10 CPython VMs.

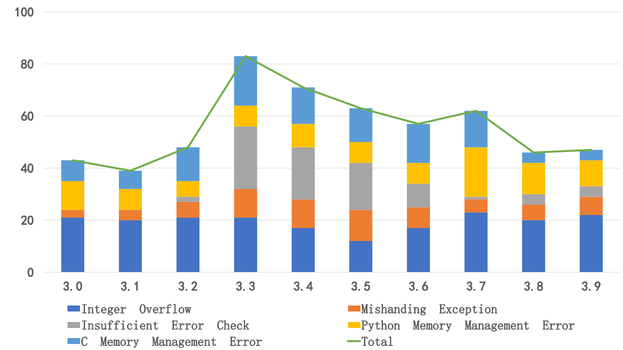

Fig. 4: CPython Vulnerabilities.

RQ1: What are the vulnerability patterns in CPython virtual machines? To answer this RQ, we conducted a manual inspection of all possible vulnerability patterns. First, we inspected the official Python/C API manual to identify possible vulnerability patterns. The patterns can be identified in this way include insufficient error checking, integer/buffer overflow, etc. Second, we studied the Python language specification thoroughly, to identify other vulnerability patterns like GIL flaws, type misuses, etc. Finally, we leveraged the bug patterns presented in existing multilingual studies [17], [23], to identify patterns like mishandling exceptions, TOCTOU and memory management flaws, etc. In our experience, to identify all these vulnerability patterns is not technically challenging but laborious, taking us two persons a month to finish.

Table II presents a summary of the results of our research effort to classify vulnerability patterns. For each vulnerability pattern, the table presents the number of vulnerabilities PyGuard identified, the static analysis tools we used to identify these vulnerabilities.

Finding 1: *We have identified 8 vulnerability patterns in CPython virtual machines, as presented in Table II. We concluded the most common vulnerabilities are integer overflows (194), C memory errors (111), and Python memory errors (99). There are few buffer overflows (9) and syntax errors (9), although the former one is more serious.*

RQ2: How do vulnerabilities evolve, among different versions of CPython virtual machines? To answer this RQ, we systematically classified the vulnerabilities found by PyGuard, grouped by CPython VM versions. In Fig. 4, we present the distribution of bugs in different versions of the Python virtual machines. Putting together the Python/C API evolution data in Table III (from [26]), we observed that the numbers of CPython vulnerabilities has increased significantly before version 3.3. The reason for this increase is the introduction of a large number of new features into Python. After version 3.3, the number of vulnerabilities has generally decreased. Taking into account the of CPython VM code size changes (Fig. 3), we can draw the conclusion that after CPython version 3.3, Python features continue to be stabilized.

Finding 2: *The CPython VMs introduce more vulnerabilities as new versions of VM are released, due to the introduction of new language features or APIs.*

RQ3: What are the most common vulnerabilities in CPython virtual machines? To answer this RQ, we have further analyzed and classified these vulnerabilities manually. By summarizing the general vulnerability patterns, we discussed mitigation for these vulnerabilities. Such mitigation provides guidance on the construction of security tools in the future.

Next, we discuss the common vulnerabilities respectively.

a) Memory management vulnerabilities: The CPython VM manages two memory regions: the native memory region and the Python memory region.

Native memory management. The CPython VM manages the native memory region via standard library functions such as `malloc` and `free`. The use of such functions can introduce vulnerabilities like dangling pointers, multiple frees, and memory leaks, etc. We use CppCheck and TscanCode to identify defects such vulnerabilities.

The following code snippet presents an example detected by PyGuard, illustrating a memory leaking vulnerability. The

```
1  // cpython-3.7/PC/bdist_wininst/install.c#
       line 2371
2  line = strdup(string);
3  keyname = strchr(line, '[');
4  if (!keyname)
5    return;
6
7  ++keyname;
8  subkeyname = strchr(keyname, ']');
9  if(!subkeyname)
10   return;
```

string copy function `strdup` allocates memory for the variable `line`. We should call the `free` function to reclaim the corresponding memory space before the return statement at lines 5 and 9, otherwise there will be a memory leak.

Python managed memory. To manage the Python memory, the Python/C interface provides three types of memory allocators: `PyMem_RawMalloc/ PyMem_RawFree`, `PyMem_Malloc/ PyMem_Free`

TABLE II: A summary of the vulnerabilities identified in CPython

Bug Patterns		Vulnerabilities	Tools Used
Integer Overflow		194	Scan scripts
Buffer Overflow		9	FlawFinder
Mishandling Exception		73	Scan scripts
Insufficient Error Check		82	Scan scripts
Memory Management Bugs	C Memory	111	CppCheck, TscanCode
	Python Memory	99	Scan scripts
Division by zero		21	FlawFinder, CppCheck, TscanCode
Syntax error		9	FlawFinder, CppCheck, TscanCode
Total		598	/

TABLE III: Evolution of the Python/C API

Version	Python/C API	Add	Remove	Change
3.2	663	179	79	30
3.3	702	115	76	2
3.4	732	30	0	16
3.5	751	19	0	2
3.6	764	13	0	9
3.7	804	43	3	6

and PyMem_MALLOC/PyMem_FREE. These APIs are called by Python to allocate and reclaim Python objects.

The following code fragment presents an example of memory leaking identified by PyGuard, due to improper uses of the Python APIs. The memory v allocated by PyMem_Malloc and PyMem_Realloc at lines 5 or 9 does not get released before the end of the function, which causes a memory leak.

```
1  // cpython-3.7/Modules/mathmodule.c#line
      1231
2  int _fsum_realloc(double **p_ptr,
      Py_ssize_t n, double *ps, Py_ssize_t *
      m_ptr){
3    if(...){
4      if(p == ps){
5        v = PyMem_Malloc(sizeof(double)*m);
6        if(v != NULL)
7        memcpy(v, ps, sizeof(double)*n);
8      }else
9        v=PyMem_Realloc(p, sizeof(double)*m);
10   }
11   if(v == NULL){
12     PyErr_SetString(PyExc_MemoryError, "
         math.fsum_partials");
13     return 1;
14   }
15   *p_ptr = (double*) v;
16   *m_ptr = m;
17   return 0;
18 }
```

Finding 3: *Memory-related vulnerabilities are still very common in CPython VMs. Although there have been a lot of researches [34]–[41] on memory management-related errors C, memory-related vulnerabilities are still difficult to locate and repair automatically.*

b) Integer Overflow: Integer overflow vulnerabilities [42] in C/C++ native code mainly include arithmetic overflow, loss of integer value conversion, and illegal use of bit operations, etc. Such vulnerabilities have drawn much research attention.

The Python virtual machine includes another type of integer overflow vulnerability that is unique to the Python/C API. In order to support the direct conversion between Python integers and C/C++ integers, the Python/C interface introduced a group of functions such as PyArg_Parse, PyArg_ParseTuple, and PyArg_ParseTupleAndKeywords. These functions use special format strings to accomplish such conversions. But misuse of these format strings can lead to integer overflows. We have systematically studied all potential unsafe format string by inspecting the language specification. In Table IV, we present format string characters and their meanings. The following code snippet gives an example of an integer overflow identified by PyGuard. The code uses the format character

TABLE IV: 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
K	integer	unsigned long long

```
1  // cpython-3.7/Modules/socketmodule.c#line
      1654
2  if(!PyArg_ParseTuple(args, "II:
      getsockaddrarg", &pid, &groups))
3    return 0;
```

without overflow checking to convert a Python integer into an unsigned int unsigned integer in C. When the integer value passed by the Python side exceeds the C unsigned int range (0-4294967295), the Python/C API will not raise an overflow

exception, but will directly truncate it, which may lead to fatal errors or exploitable vulnerabilities.

Finding 4: *There are integer overflow vulnerabilities in the CPython virtual machines. Such vulnerabilities can be repaired in a syntax-directed and automatic manner.*

c) Mishandling Exceptions: CPython provides a group of API functions such as `PyErr _SetString` and `PyErr _SetObject` to raise Python exceptions. However, there is a mismatch between the Python exception handling mechanism and the Python/C API exception handling mechanism. When an exception is thrown in the Python code, the Python interpreter will terminate it immediately and pass control to the nearest try statement. The exception raised by the Python/C API cannot immediately terminate the execution of the native method, and must be explicitly returned to avoid unexpected control flow.

The following code fragment present a mishandling exception reported by PyGuard. The fix for this code is very simple,

```
1  // cpython-3.7/Objects/genobject.c#line 369
2  if(err == 0)
3   PyErr_SetNone(PyExc_GeneratorExit);
4  retval = gen_send_ex(gen, Py_None, 1, 1);
```

we should add a return statement after line 4. But for function calls, the situation becomes more complicated, a caller must handle two cases explicitly: the callee returns normally, or the callee returns abnormally by throwing an exception.

d) Insufficient Error Check: The C language does not have an exception mechanism, so it is necessary for the caller to perform an explicit check on the return value to process errors returned by the callee. If the caller ignores the check of the return value, there may be vulnerabilities.

The following code fragment illustrates a vulnerability of insufficient error check reported by PyGuard. Before execut-

```
1.// cpython-3.7/Python/bltinmodule.c#line 2433
2.result = PyFloat_FromDouble(f_result);
3.if(result != NULL){
4.    ...
5.    temp = PyNumber_Add(result, item);
6.    ...
7.}
```

ing `PyNumber_Add`, this code should first check whether `result` is non-empty (line 3), otherwise it will lead to a security hole.

V. Discussion

In this section, we discuss the limitation of this work, and also present future research directions. It should be noted that this work represents the first step towards finding and understanding vulnerabilities in real-world Python virtual machines, by constructing a new software prototype PyGuard.

Static analysis is useful for detecting and locating security bugs, as demonstrated by the number of bugs we identified with the help of static analysis tools. On the other hand, there are a few limitations of the current state-of-the-art of static analysis tools.

The tools we used issued a large number of warnings that are false positives. For each of the three off-the-shelf tools, Table V lists the number of warnings it issued, the number of true errors, and its false-positive rate.

TABLE V: False positive rate of static analysis tools.

Static analysis tools	Warnings	Errors	FP rates
FlawFinder	18913	37	99.8%
CppCheck	698	54	92.3%
TscanCode	1203	74	93.8%

Our own algorithms and scanners perform slightly better, but the false-positive rates are still high; as illustrated by Table VI.

TABLE VI: False-positive rates of our tools.

Bug Patterns	Warnings	Errors	FP rates
Integer Overflow	450	194	56.9%
Mishandling Exception	2297	73	96.8%
Insufficient Error Check	2491	82	96.7%
Memory management flaws (Python Memory)	822	99	87.9%

Flawfinder, CppCheck, TscanCode and our own tools are based on simple syntactic pattern matching. As such matchings are coarse-grained, they all have high false positive rates. We believe false-positives rates can be significantly reduced through the use of advanced static techniques such as software model checking (e.g., MOPS [54], CMC [56], SLAM [53]), and type qualifiers [57], [58], theorem proving techniques (e.g., ESC/Java [55]), among others.

In addition, it is worth further exploring the vulnerability analysis technology of Python virtual machine native methods, based on dynamic scanning such as symbol execution [49]–[51] and blot analysis [52].

Finally, as vulnerability modeling and vulnerability discovery technology based on machine learning [43]–[48] are becoming more and more promising, it's worthy to explore the techniques to scan and analyze Python virtual machine code using machine learning technologies.

VI. Conclusion

The work in this paper present a novel software prototype PyGuard to scan vulnerabilities in Python virtual machines, and to understand the vulnerability patterns. With PyGuard, we carried out an empirical study of 10 different versions of CPython virtual machines. By scanning a total of 3,358,391 lines native code, we have identified 598 new vulnerabilities.

Based on our research results, we propose a taxonomy to classify vulnerabilities in CPython virtual machines, and suggest systematic remedies that can mediate the threats posed

by these vulnerabilities. Our taxonomy provides a guidance to construct automated and accurate bug-finding tools for Python virtual machines.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their valuable feedback. This work is supported by a graduate education innovation program of USTC under grant No. 2020YCJC41.

REFERENCES

[1] Type system. https://en.wikipedia.org/wiki/Type_system#Type_safety_and_memory_safety.
[2] Jim Blandy, Jason Orendorff. Programming-Rust, Chapter 1. Why Rust?-Type Safety. 18-20.
[3] PATRICIA JOHNSON: What Are The Most Secure Programming Languages. WhiteSource, MARCH 17, 2019.
[4] CVE-2021-23336. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-23336.
[5] CVE-2018-1000117. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1000117.
[6] CVE-2017-1000158. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1000158.
[7] CVE-2016-5636. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5636.
[8] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, Westley Weimer: CCured: type-safe retrofitting of legacy software. ACM Trans. Program. Lang. Syst. 27(3): 477-526 (2005).
[9] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, Yanling Wang: Cyclone: A Safe Dialect of C. USENIX Annual Technical Conference, General Track 2002: 275-288.
[10] Zhilong Wang, Xuhua Ding, Chengbin Pang, Jian Guo, Jun Zhu, Bing Mao: To Detect Stack Buffer Overflow with Polymorphic Canaries. DSN 2018: 243-254.
[11] Young-Su Jang, Jin-Young Choi: Automatic Prevention of Buffer Overflow Vulnerability Using Candidate Code Generation. IEICE Trans. Inf. Syst. 101-D(12): 3005-3018 (2018).
[12] Jiadong Ren, Zhangqi Zheng, Qian Liu, Zhiyao Wei, Huaizhi Yan: A Buffer Overflow Prediction Approach Based on Software Metrics and Machine Learning. Secur. Commun. Networks 2019: 8391425:1-8391425:13 (2019).
[13] William Arild Dahl, Laszlo Erdodi, Fabio Massimo Zennaro: Stack-based Buffer Overflow Detection using Recurrent Neural Networks. CoRR abs/2012.15116 (2020).
[14] Gregory J. Duck, Roland H. C. Yap: EffectiveSan: Type and Memory Error Detection using Dynamically Typed C/C++. CoRR abs/1710.06125 (2017).
[15] Hang Xu, Wei Ren, Zimian Liu, Jiajun Chen, Junhu Zhu: Memory Error Detection Based on Dynamic Binary Translation. ICCT 2020: 1059-1064.
[16] Wenjie Li, Dongpeng Xu, Wei Wu, Xiaorui Gong, Xiaobo Xiang, Yan Wang, Fangming gu, Qianxiang Zeng: Memory access integrity: detecting fine-grained memory access errors in binary code. Cybersecur. 2(1): 17 (2019).
[17] Michael Furr, Jeffrey S. Foster: Checking type safety of foreign function calls. PLDI 2005: 62-72.
[18] Michael Furr, Jeffrey S. Foster: Polymorphic Type Inference for the JNI. ESOP 2006: 309-324.
[19] Gang Tan, Andrew W. Appel, Srimat Chakradhar, Anand Raghunathan, Srivaths Ravi, Daniel Wang: Safe Java Native Interface. IEEE International Symposium on Secure Software Engineering, 2006: 97¨C106.
[20] Siliang Li, Gang Tan: Finding bugs in exceptional situations of JNI programs. CCS 2009: 442-452.
[21] Siliang Li, Gang Tan: JET: exception checking in the Java native interface. OOPSLA 2011: 345-358.
[22] Siliang Li, Gang Tan: Exception analysis in the Java Native Interface. Sci. Comput. Program. 89: 273-297 (2014).
[23] Gang Tan, Jason Croft. An Empirical Security Study of the Native Code in the JDK. USENIX Security Symposium 2008: 365-378.
[24] Siliang Li, Gang Tan. Finding Reference-Counting Errors in Python/C Programs with Affine Analysis. ECOOP 2014: 80-104.
[25] Junjie Mao, Yu Chen, Qixue Xiao, Yuanchun Shi. RID: Finding Reference Count Bugs with Inconsistent Path Pair Checking. ASPLOS 2016: 531-544.
[26] Mingzhe Hu, Yu Zhang. The Python/C API: Evolution, Usage Statistics, and Bug Patterns. SANER 2020: 532-536.
[27] Python Developers Survey 2020 Results. https://www.jetbrains.com/lp/python-developers-survey-2020/.
[28] WHEELER, D. A. Flawfinder. http://www.dwheeler.com/flawfinder/.
[29] Daniel Marjam?ki. Cppcheck. http://cppcheck.sourceforge.net/.
[30] Tencent. TscanCode. https://github.com/Tencent/TscanCode.
[31] Coverity. https://scan.coverity.com/.
[32] Checkmarx. https://www.checkmarx.com/.
[33] Fortify. https://www.joinfortify.com/.
[34] Piyus Kedia, Manuel Costa, Matthew J. Parkinson, Kapil Vaswani, Dimitrios Vytiniotis, Aaron Blankstein: Simple, fast, and safe manual memory management. PLDI 2017: 233-247.
[35] Matthew Davis: Automatic memory management techniques for the go programming language. University of Melbourne, Australia, 2015.
[36] Liviu Codrut Stancu: Safe and Efficient Hybrid Memory Management for Java. University of California, Irvine, USA, 2015.
[37] J. Baltasar García Pérez-Schofield, Matías García-Rivera, Francisco Ortin, María J. Lado: Learning memory management with C-Sim: A C-based visual tool. Comput. Appl. Eng. Educ. 27(5): 1217-1235 (2019).
[38] P. Pufek, H. Grgic, B. Mihaljevic: Analysis of Garbage Collection Algorithms and Memory Management in Java. MIPRO 2019: 1677-1682.
[39] Maroua Maalej, S. Tucker Taft, Yannick Moy: Safe Dynamic Memory Management in Ada and SPARK. Ada-Europe 2018: 37-52.
[40] Vasilios Kelefouras, Karim Djemame: A methodology for efficient code optimizations and memory management. CF 2018: 105-112.
[41] Martin Holm Cservenka, Robert Glück, Tue Haulund, Torben Ægidius Mogensen: Data Structures and Dynamic Memory Management in Reversible Languages. RC 2018: 269-285.
[42] Will Dietz, Peng Li, John Regehr, Vikram S. Adve: Understanding Integer Overflow in C/C++. ACM Trans. Softw. Eng. Methodol. 25(1): 2:1-2:29 (2015).
[43] Zeki Bilgin, Mehmet Akif Ersoy, Elif Ustundag Soykan, Emrah Tomur, Pinar ?omak, Leyli Kara?ay: Vulnerability Prediction From Source Code Using Machine Learning. IEEE Access 8: 150672-150684 (2020).
[44] Oleksandr A. Letychevskyi, Yaroslav Hryniuk: Machine Learning Methods for Improving Vulnerability Detection in Low-level Code. IEEE BigData 2020: 5750-5752.
[45] Shivi Garg, Niyati Baliyan: Machine Learning Based Android Vulnerability Detection: A Roadmap. ICISS 2020: 87-93.
[46] Jian Jiang, Xiangzhan Yu, Yan Sun, Haohua Zeng: A Survey of the Software Vulnerability Discovery Using Machine Learning Techniques. ICAIS (4) 2019: 308-317.
[47] Boris Chernis, Rakesh M. Verma: Machine Learning Methods for Software Vulnerability Detection. IWSPA@CODASPY 2018: 31-39.
[48] Jacob A. Harer, Louis Y. Kim, Rebecca L. Russell, Onur Ozdemir, Leonard R. Kosta, Akshay Rangamani, Lei H. Hamilton, Gabriel I. Centeno, Jonathan R. Key, Paul M. Ellingwood, Marc W. McConley, Jeffrey M. Opper, Sang Peter Chin, Tomo Lazovich: Automated software vulnerability detection with machine learning. CoRR abs/1803.04497 (2018).
[49] Kasper Se Luckow, Rody Kersten, Corina S. Pasareanu: Complexity vulnerability analysis using symbolic execution. Softw. Test. Verification Reliab. 30(7-8) (2020).
[50] Juraj Vijtiuk, Luka Perkov, Antonio Krog: Bug detection in embedded environments by fuzzing and symbolic execution. MIPRO 2020: 1218-1223.
[51] R¨¦ka Kov¨¢cs, G¨¢bor Horv¨¢th, Zolt¨¢n Porkol¨¢b: Detecting C++ Lifetime Errors with Symbolic Execution. BCI 2019: 25:1-25:6.
[52] Malte Mues, Till Schallau, Falk Howar: Jaint: A Framework for User-Defined Dynamic Taint-Analyses based on Dynamic Symbolic Execution of Java Programs. Software Engineering 2021: 77-78.
[53] Thomas Ball, Rupak Majumdar, Todd D. Millstein, Sriram K. Rajamani: Automatic Predicate Abstraction of C Programs. PLDI 2001: 203-213.
[54] Hao Chen, David A. Wagner: MOPS: an infrastructure for examining security properties of software. CCS 2002: 235-244.
[55] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, Raymie Stata: PLDI 2002: Extended static checking for Java. ACM SIGPLAN Notices 48(4S): 22-33 (2013).

[56] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, David L. Dill: CMC: A Pragmatic Approach to Model Checking Real Code. OSDI 2002.
[57] Jeffrey S. Foster, Tachio Terauchi, Alexander Aiken: Flow-Sensitive Type Qualifiers. PLDI 2002: 1-12.
[58] David Greenfieldboyce, Jeffrey S. Foster: Type qualifier inference for java. OOPSLA 2007: 321-336.