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 of 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 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 has 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 evolutions 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.

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/
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>
<thead>
<tr>
<th>Module</th>
<th>Files of Native Code</th>
<th>Lines of Native Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doc</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Grammar</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Include</td>
<td>99</td>
<td>16442</td>
</tr>
<tr>
<td>Lib</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Misc</td>
<td>1</td>
<td>187</td>
</tr>
<tr>
<td>Modules</td>
<td>108</td>
<td>189728</td>
</tr>
<tr>
<td>Objects</td>
<td>42</td>
<td>88515</td>
</tr>
<tr>
<td>Parser</td>
<td>19</td>
<td>5259</td>
</tr>
<tr>
<td>PC</td>
<td>20</td>
<td>8985</td>
</tr>
<tr>
<td>Programs</td>
<td>3</td>
<td>503</td>
</tr>
<tr>
<td>Python</td>
<td>70</td>
<td>78001</td>
</tr>
<tr>
<td>Tools</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>362</strong></td>
<td><strong>387620</strong></td>
</tr>
</tbody>
</table>

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.
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 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.

Finding 3: 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.

Python managed memory. To manage the Python memory, the Python/C interface provides three types of memory allocators: PyMem_RawMalloc/PyMem_RawFree, PyMemMalloc/PyMemFree
Table II: A summary of the vulnerabilities identified in CPython

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

Table III: Evolution of the Python/C API

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

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 in 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 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.

```
1  // cpython-3.7/Modules/socketmodule.c#line 1654
2  if(!PyArg_ParseTuple(args, "II: getsockaddrarg", &pid, &groups))
3      return 0;
```
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, 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 executing PyNumber_Add, this code should first check whether result is non-empty (line 3), otherwise it will lead to a security hole.

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);

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.

<table>
<thead>
<tr>
<th>Static analysis tools</th>
<th>Warnings</th>
<th>Errors</th>
<th>FP rates</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlawFinder</td>
<td>18913</td>
<td>37</td>
<td>99.8%</td>
</tr>
<tr>
<td>CppCheck</td>
<td>698</td>
<td>54</td>
<td>92.3%</td>
</tr>
<tr>
<td>TscanCode</td>
<td>1203</td>
<td>74</td>
<td>93.8%</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Bug Patterns</th>
<th>Warnings</th>
<th>Errors</th>
<th>FP rates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Overflow</td>
<td>450</td>
<td>194</td>
<td>56.9%</td>
</tr>
<tr>
<td>Mishandling Exception</td>
<td>2297</td>
<td>73</td>
<td>96.8%</td>
</tr>
<tr>
<td>Insufficient Error Check</td>
<td>2491</td>
<td>82</td>
<td>96.7%</td>
</tr>
<tr>
<td>Memory management flaws (Python Memory)</td>
<td>822</td>
<td>99</td>
<td>87.9%</td>
</tr>
</tbody>
</table>

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], 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 blox 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. 2020YJCIC41.

REFERENCES


