2012 IEEE International Conference on Information Science and Technology
Wuhan, Hubei, China; March 23-25, 2012

PyXhon: Dynamic Detection of Security Vulnerabilities in Python

Ming Sun, Dawu Gu, Juanru Li, and Bailan Li

Abstract— **Python programming language supports third-party software extensions which are important for software prototype development. This paper presents a security enhancement plug-in PyXhon, that detects the security vulnerabilities and privacy leaks from third-party extensions. We propose the *Function Oriented Analysis*, which developers use to monitor all function-call procedures; dynamic *Byte Instruction Trace Analysis*, which infers the behaviors of importing modules and accessing private DLL; and *security policies*, which provides strategies to accept or reject extensions. These security mechanisms do not require Python language features so as to be completely transparent to Python applications. PyXhon could generate a violation report, which helps developers quickly locate and analyze suspect code of extensions. To demonstrate the usefulness of PyXhon, we have analyzed more than 30 popular Python third-party extensions. Our experiments show that, with the violations of some extensions, most third-party code respect the resources privilege.**

I. Introduction

SCRIPTING languages are more conventional programming languages and are widely used in text processing, scientific computing and web development. Scripting languages usually allow users to extend the functionality via third-party specific software packages and take advantage of the sufficient third-party extensions. However, the security research about those third-party extensions has not been well considered in both academic and engineering fields.

Third-party code brings more difficulties of ensuring the software security. Even if third-party extensions are signed and published by the trusted groups, it is likely that there are more security issues and security vulnerabilities than the well-considered underlying platform. Many third-party extensions are not developed and tested sufficiently because of no-commercial delivery or limited human resource; they may also be ignorant of the implicit features in different platforms. In addition, third-party extensions usually collect the personal information in order to count and analyze users' profiles.

There are many measures and policies to guard against the third-party malicious code in many programming languages, such as Java and Flex[2]. Java security model is based on customizable *sandbox* in which Java programs can run safely without potential risk to system, while Flex acquires the system permission before accessing sensitive resources. However, Flex will make the resource accessible regardless of how programs manipulate resources after permissions are granted.

Python[1] is an interpreted high-level programming language whose design emphasizes on code readability and functional extensibility. Python has definite standard libraries, usually cited as the basic features of Python language, providing the common tools and algorithms. The programming philosophy of Python is to use appropriate third-party extensions instead of implementing the functionality by developers themselves. Those internal functions provide developers sufficient flexible tricks while it is convenient for adversaries to download the malicious code from Internet and execute them immediately. It is difficult for existing Python security projects to capture whole potential security vulnerabilities and privacy risks in third-party extensions. Indeed, novel Python analysis approach is necessary to trace the complete information flow and evaluate the dynamic behavior before adopting the extensions.

Currently Python has many extensible and dynamic features, such as internal function *execfile()* and *exec()*, which bring Python users numerous potential risks. At the same time, Python lacks mechanisms for controlling access to resources so that it is hard to control the Python execution within *sandbox* like Flex. Then malicious code can access the operating system APIs by importing internal *ctypes* library, which may harm the availability and confidentiality of system. It is difficult to guide system away from leaking personal information, but in scenarios, the unexpect behaviors must be concerned by users who maintain the security of confidential data.

This paper proposes PyXhon, a prototype security enhancement of Python interpreter to reveal the malicious software behaviors. PyXhon does not prevent third-party code execution by restricting their access to private resources or removing specified modules in Python language, which indeed change Python language characteristics. The primary goal of PyXhon is to evaluate third-party extensions by monitoring the function-call procedures and analyzing byte instructions traces so as to figure out the whole malicious behaviors and potential security risks.

II. Related Works

This paper's focus is the intersection of third-party extensions and Python interpreter. There are several existing security enhancement approaches for scripting language, which is either to restrict what modules are available in

This work was supported by China Major National S&T Program (2012ZX03002011) and Shanghai Municipal Commission of Economy and Information Major High-Tech Program.

Ming Sun, Dawu Gu, Juanru Li and Bailan Li are with Department of Computer Science and Engineering of Shanghai Jiao Tong University, Shanghai, China (email: {sterling.sun, dwgu, jarod, lbl}@sjtu.edu.cn).

Python interpreter or to disallow the tainted data accessing the information flow.

InvisiType[3] provides object-oriented security check based on the fundamental hierarchies in Python object-oriented paradigm. Solutions in *InvisiType* are to encapsulate the safety checks in an external Python library, which developers must introduce to gain fine control over third-party extensions as security policies. Taint analysis is introduced to enhance the Python's built-in information flow as taint mode Python library[9]. There are also many other external libraries[7][14] used to evaluate the third-party extensions security. Most of libraries require developers to follow their criteria and policies so that it is difficult to adopt them to existing projects. Several engineering projects provide the modified Python interpreter whose dynamic modules with potential risks are removed, such as GAE[10] and Simple Interpreter[6]. Those approaches do change the Python language features and violate the Python operating mechanisms. If there are trivial violations in the third-party extensions, it is impossible to evaluate and exploit third-party extensions because they cannot be executed anymore without critical modules. We must employ those functional extensions at some situations, but those downsizing Python are bound to let developers down.

There are some static analysis methods[8] to evaluate the security of third-party extensions. By parsing the Python source code into abstract syntax trees (ASTs), static analysis is to verify the potential work flow or forbid accessing high sensitive internal functions and modules. Unfortunately, static evaluation approach cannot tackle dynamic features in Python, like the obfuscated code and the encrypted data. RESIN[4] is a PHP runtime that helps prevent security vulnerabilities by allowing developers to specify application-level data flow assertion. RESIN's idea is effective for Web application while it is not sufficient for the evaluation of third-party extensions.

Our work is quite different with traditional researches about Python safety execution which is mainly concerned with controlling access to the Python execution. PyXhon is to evaluate whole potential security issues in third-party extensions by monitoring the specified function-call procedures and module-importing actions in Python interpreter, which is completely transparent to Python applications and provides accurate analysis without reading source code line by line.

III. Threat Model

The purpose of the paper under discussion is to handle a threat model where the third-party Python extensions are running in the context of python interpreter without consideration security of external Dynamic Link Library (DLL). Python is an open source programming language then it is easy to parse source code to ASTs and analyze the control flow according to semantics. Adversary can also have a thorough understanding about Python language and make use of tricky mechanisms like *ctypes* module and obfuscation technology to protect their malicious code.

We should suppose that adversary has sufficient knowledge about the current analyzing technologies and good understanding of the popular security-enhancement projects. It is also convenient for adversary to detect whether there are well-known security libraries in interpreter context. With those kinds of smart-adversary considered, the best way to evaluate third-party code is to dynamically monitor how the interpreter works, which is totally transparent to high-level programming language. Any code-protection scheme based on Python language cannot infer the difference between original interpreter and the one with PyXhon.

PyXhon assumes there are not malicious code in Python internal libraries, and does not prevent an adversary from compromising the underlying middleware and Operating System. We also do not consider the situation that OS may restrict the resource accessing upon individual processes as system security policies. The most primitive mean of evaluating extensions is to line-by-line analyze source code, which is bound to be ineffective and costly. Automatic analyzing solutions are urgent to be implemented to benefit developers. PyXhon is one kind of those solutions to evaluate the behaviors of third-party extensions.

IV. Design

Many projects described in Section 2 can tackle the third-party executions by forbidding accessing specified resources and blocking the suspect code, but the design of such Python library cannot be transparent to the third-party code because they may pollute the Python context. Instead of preventing the violation code execution, our goal is to total transparently monitor the behaviors of third-party code and evaluate their vulnerabilities and privacy risks. This section describes how PyXhon addresses the malicious and privacy risks in third-party, and presents Function Oriented Analysis (FOA) approach, dynamic Byte Instruction Trace Analysis (BITA) approach and PyXhon security policy.

A. Overview

To illustrate the high-level design of PyXhon and what developers must to do before evaluating third-party extensions, this part describes the foundational knowledge about how Python Interpreter works, especially internal different function-call procedure and dynamic byte instruction trace. And it is essential to define the policies that tell the developers which function-call does not satisfy security policies and how byte instruction flow is unauthorized. There are three components in PyXhon plug-in, which are function oriented analysis module, dynamic byte instruction analysis module and policy configuration module. The structure of PyXhon is shown in Figure 1.

B. Function Oriented Analysis

PyXhon aims to monitor all function-call procedures in Python interpreter, which can reveal the reappearance of program control flow and data flow. Generally speaking, there are two kinds of function paradigms, the procedure function and object-oriented method.

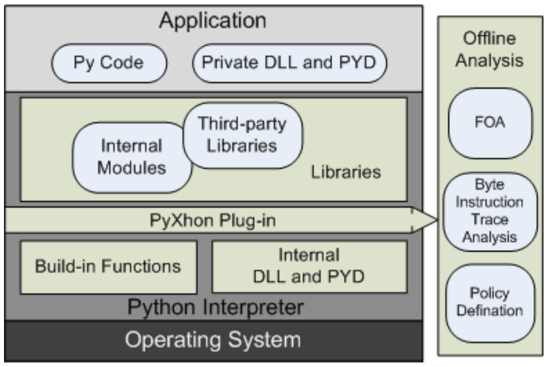

Fig. 1. Architecture of PyXhon Plug-in

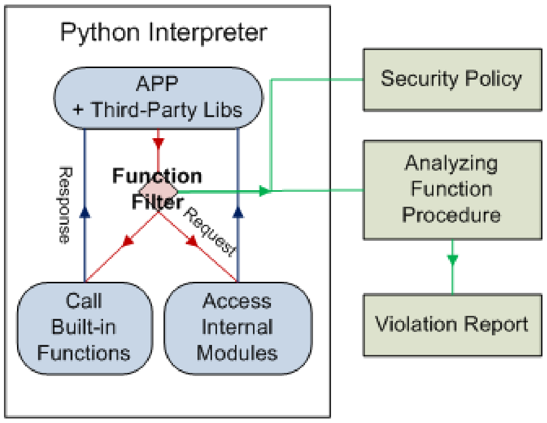

Fig. 2. PyXhon Function Oriented Analysis Workflow

Python interpreter has a number of procedure functions built into it that are always available, such as *open()*, *execfile()*. To facilitate developers and accelerate software development, those built-in functions play important role. However, built-in functions have advanced features like *execfile()* which can compile and execute text files from Internet or other untrusted source. Abuse of those functions will bring the potential system risk and privacy leak, including deleting critical files, destroying the system availability and sending confidential information via Internet.

Python also supports object-oriented paradigm which wrap up methods and data with standard class. It is common practice to separate the source code into different modules, and Python language follows this practice with many utility internal libraries, most of which are embracing the object-oriented paradigm. The member function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions of methods. Python interpreter can easily distinguish the member function call and the relative argument list because it is by the name of member function that interpreter can find the corresponding function object in context. Like procedural function, member function of object-oriented programming can also identify execution functionality by different function names and various argument lists.

PyXhon provides novel FOA approach, which can discover the critical function-call in Python programs transparently. FOA tracks both internal and external functions and relative argument list which determines the expected consequences. PyXhon will focus on and record those specific and critical functions, depending on the policy configuration. If developers want to check whether third-party extensions will send private data to untrusted server, they can monitor the *send()* function in *socket.py* module. If developers want to verify whether the Python code has accessed system files, it is convenient to monitor the built-in function *execfile()*, *open()* and *file()*, because Python developers will not overwrite those internal key functions.

C. Dynamic Byte Instruction Trace Analysis

When developers invoke the Python interpreter, the source code is scanned for lexical analysis which transforms the plain text into tokens. Then these tokens are parsed into ASTs representing the logical structure of source code as the Python grammar description. The code generation phase of compilation take the ASTs generated into *PyCodeObject* which are independent units of executable code, containing all the byte code and relative data structure for Python interpreter. The execution of byte instruction is handled in the Python interpreter, translating byte instruction to machine code with manipulating the Python context stack.

The trace of Python byte instructions and operands could represent all characteristics of corresponding Python source code, because this trace contains all the data manipulation process and relative values. The byte instructions can provide developers some important clues, such as byte instruction *CALL_FUNCTION* to destine the function-call procedure, *IMPORT_FROM* to push attributes from one module onto the stack. However, it is difficult to infer complete appearance of Python source from byte instruction flow, because low-level instructions are not sufficient and hard to reappear the runtime Python stack.

There are many researches about the dynamic binary flow[13] on X86 architecture, include detecting the sensitive data pattern of the instructions of cryptographic primitives, secret keys and system call procedures. This analyzing idea can be applied to Python byte instruction analysis, detecting the resource acquisition and data operation. In order to improve the efficiency and flexibility of BITA, PyXhon intends to offline analyze the byte instruction trace and associated operation data flow. Currently, PyXhon only presents the analysis for importing libraries and accessing DLL functions. Other further inferences from dynamic byte instruction trace are not implemented and will be well analyzed in the future.

D. Security Policy

Measures used today to prevent against the malicious code from third-party extensions are very crude, which only forbid accessing the identified functions and isolate the certain system resources. Those rude approaches cannot objectively evaluate third-party extensions because in particular situations it is reasonable to access the critical built-in functions

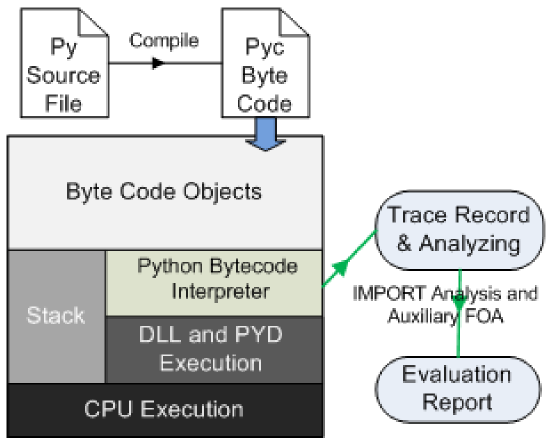

Fig. 3. PyXhon Dynamic Byte Instruction Trace Analysis

and system services. PyXhon provides sophisticated Python third-party security policies, help developers to employ user-desired features in extension libraries, under the promise of fine-grained security policies.

By FOA and BITA, PyXhon could recognize the whole function-call procedures and module importing operations. It is necessary to provide the criterion to distinguish the well-performance third-party code against the malicious one. We propose a precious policy which states that third-party code are not allowed to access the independent resources. In other words, a third-party extension for text processing is unreasonable to import the socket module, otherwise PyXhon security policy will report this violation. Generally, we can avoid the abuse of system resources by function categories policy, which are based on the different major functionality of built-in functions and internal libraries. There are five basic functionality categories according to our policy definition in Table I.

Our strategies are totally different from the sandbox security evaluation approach and restricted execution mode, but to provide the independent runtime environment for third-party extensions. As monitoring how interpreter works, PyXhon does not depend on any Python language feature. This allows developers to customize precious security policies, to monitor procedural functions in any module or member functions in all classes. However, PyXhon cannot tackle the malicious DLL in the Python interpreter level, it is another security topic about the binary DLL reverse engineering.

V. Implementation

We have implemented PyXhon prototype in CPython runtime, version 3.1.2, as Python3 has significant difference with the previous versions, designed more graceful and flexible. To be flexible and support experimentation, the dynamic Byte Instruction Trace Analysis module and Evaluation Report module are written in Python itself. Instead of removing the particular built-in functions and internal libraries, PyXhon solution keeps all the features in Python language without restricting any resources.

PyXhon takes full advantage of the idea of Aspect-Oriented Programming, adding configurable filter in function-call aspect and byte instruction dispatching section in Python interpreter. First, Python prototype stores the dynamic function information and byte instruction traces in specified output files, named *BytecodeFlow*, *FunctionInfo* and *ImportModule*; it is an offline analysis strategy on behalf of execution efficiency. Second, PyXhon will parse these data-set and give analysis conclusion according to the security policies.

To allow developers to customize the security policies, as described in Section4.4, PyXhon provide the practical configuration file, the interface to change the different policies and monitor the specified functions. For instance, when evaluating a third-party network enforcement library, as Python wrapper around Twitter API, it is convenient to define the functionality policy that this extension can accesses Network Communication Category and other module resources access are illegal. Furthermore, we can also define the more fine-grained policies, which only allow extensions access to particular functions and classes in the Network Communication Category. In PyXhon prototype, it is important that developers should define the appropriate policies upon type and usage of the third-party extension.

VI. Evaluation

The main criteria for evaluating PyXhon is whether it is effective at helping developers diagnose the security violations in third-party extensions. PyXhon could automatically analyze the critical areas in extensions, which is more efficient and precious than the manual analysis.

To determine the capability and precision of PyXhon, we evaluated many wide-used, famous third-party extensions of Python3 and figured out violations according to our security policies. Table II summarizes the result of some typical extensions, showing the extension name, the specified extension security policies and the violations in extensions. Developers should review source code in the extension according to the evaluation report, to determine whether those violations indeed have security vulnerabilities and privacy leaks.

A. *Experimental Results*

We have chosen various widely-used Python third-party extensions[14], latest distribution from the official websites. There is the precondition that high-level functionality of the extensions is known to the users. We can easily implement test programs based on the interface of third-party extensions. Then PyXhon will evaluate test programs to analyze the behaviors of adopted third-party extensions.

Some extensions are native Python code while the others are wrappers of the current services and libraries that are implemented in another programming language. There are different policies for native code and wrappers, because wrappers mostly depend on various private DLL and PYD. Security policy should be defined according to the functionality of the adopted part in extensions. For instance, policies of scientific computing extension *numpy* should

TABLE I
CATEGORY OF INTERNAL FUNCTIONS AND MODULES IN PYTHON 3.1.2

Categories Policy	**Functions**	**Modules**
File Operation	- file, open, execfile	- fileinput, filecmp, linecache, shutil, glob, fnmatch, tempfile, shelve, marshal, dbm, sqlite3, zlib, gzip, bz2, zipfile, tarrfile, csv, configparser, metrc, xdrlib, plistlib, io, etc.
Network Communication	- N/A	- io, socket, _socket, ssl, _ssl, asyncore, asynchat, email, mailcap, mailbox, webbrower, cgi, wsgiref, urllib, http, ftplib, poplib, imaplib, nntplib, smtplib, smtnd,socketsever, xmlrpc, etc.
Local DLL Access	- __import__	- ctypes, _ctypes, import
Operating System Service	- N/A	- ctypes, sys, os, platform, subprocess, ossadiodev, cmd, msilib, msvcrt, winreg, winsound, _mis, posix, grp, crypt, termios, tty, pty, fcntl, resource, nis,pwd, spwd, etc.
Dynamic Built-in Functions	- eval, exec, execfile, compile, locals,globals, memoryview, __import__	- codeop, py_compile, compileall

TABLE II
TYPICAL RESULTS FROM USING PYXHON TO EVALUATE THIRD-PARTY EXTENSIONS

	Policy	File(F)	Network(N)	DLL(D)	OS(O)	Dy Built-ins(B)	Suggestion
Numpy 1.6.0	D-O-B	-	-	Y	Y	Y	Accept
Cx_Freeze 4.2.2	F-O-B	Y	Violation(1)	-	Y	Y	CodeReview
Pyisbn	None	-	Violation(1)	-	-	-	CodeReview
Pyro 4.4.6	N-O-B	-	Y	Violation(1)	Y	-	CodeReview
MDP 3.1	FDOP	Y	-	*import ctypes* in Numpy	Y	Y	Accept
crcmod 1.7	D-O	-	-	Y	Y	-	Accept
xlrd3 0.1.4	F-O	Y	-	-	Y	-	Accept

allow the *Local DLL Access*, *Dynamic Built-in Functions* and *Operating System Call*(D-O-B), but not allow *File Operation* and *Network Communication*, because it is not reasonable for *numpy* to access the file system and network interface. When unexpected functions appear, there may be a security violation in third-party code. Then developers could manually check the corresponding source code easily because of the violation location information, provided by PyXhon. This approach improves the efficiency and precision of the security checking, as many investigation works are done by PyXhon for developers. Finally, developers just need focus their attention on critical code in extensions.

We demonstrate the capability of PyXhon by evaluating the different types of Python extensions, which are widely used in Python software development. The results in Table II indicate that PyXhon can detect the potential vulnerabilities quickly even if vulnerabilities hide themselves in obfuscation code of extensions. For instance, *cx_Freeze* library is a set of scripts and modules for freezing Python scripts into executables, whose major work is to extract the necessary resources and make them into a package. So it is not necessary to access the Network Communication and Private DLL belonging to Python internal resources. PyXhon found that there is *import socket* instruction execution and the *socket.gethostname()* function-call procedure during execution. As our security policies, those unexpected importing and function-call events are violations so that the evaluation report must provide the locations and arguments information of the violation. Developers must check the relative source code of the extension, according to PyXhon evaluation report. *Pyisbn* is an extension used to calculate of ISBN checksums, but it may bring us potential risks because adversaries could take advantage of the abuse use of email module in *Pyisbn* to leak personal information. This suggests that when developers want to take advantage of one external extension, they must evaluate the security issues before distributing applications to customer.

B. Generality and Performance

What make PyXhon unique is that developers could evaluate the extensions without consideration the code protection technologies in third-party code. PyXhon is implemented in Python interpreter, completely transparent to the Python programming language so that it is immune from the language-based code obfuscation and encryption. The adversary cannot exploit the vulnerabilities of PyXhon by inserting the malicious code to destroy the security mechanism. The results of above experiments also show that PyXhon could preciously demonstrate the behaviors of extensions. The idea of FOA can also be applied to other scripting languages, such as Perl, PHP and JavaScript.

Although the main focus of PyXhon is to detect the security vulnerability of third-party extensions, developers may reject this approach if they impose excessive performance overhead. We also measured the overhead during evaluation, showing that PyXhon brought about 20% overheads in average. However, those overhead cannot be brought into the execution of Python applications, because developers only use PyXhon in extension evaluation phases.

VII. LIMITATIONS AND FUTURE WORK

PyXhon is not a one-size-fits-all solution to detect the violation code in the third-party extensions. There are also some limitations in current PyXhon, which we will address in the future work. First, it is impossible for PyXhon to tackle the vulnerabilities in private DLL. The adversary could insert malicious code into its DLL so as to escape the security check mechanism in Python interpreter. In order to determine the security of associated DLL, we would like to integrate the excellent DLL analysis tools into PyXhon, as to be a complete Python security-enhancement system. Second, PyXhon have implemented the inference for behaviors of importing modules and accessing private DLL. Because dynamic byte instruction traces have sufficient execution information, indicating the security issues, further research about traces will continue. Finally, we will improve the efficiency of PyXhon so that there are no obvious differences between original interpreter and PyXhon. Then PyXhon could be a realtime security monitor in Python interpreter, to tackle unexecuted code of extensions in evaluation phases.

VIII. Conclusion

Our approach, PyXhon, employs the novel Function Oriented Analysis and dynamic Byte Instruction Trace Analysis, can automatically provides developers analysis report to help detecting the malicious code and privacy risk. PyXhon also provides the policy configuration interface, so that developers could apply different security policies as the various functionality of extensions. Finally, developers could make their decision by checking the violation code according to the PyXhon security evaluation report.

We also evaluated the capability of PyXhon by detecting violations in existing Python third-party extensions as corresponding security policies. Results show that PyXhon is effective at evaluating the potential vulnerabilities and privacy risks without consideration of Python code protection techniques. We hope our ideas will help Python developers prevent their applications from the risks in third-party extensions.

References

[1] Python Documentation. Available: http://www.python.org/doc.

[2] Flex. Available: http://www.adobe.com/products/flex/.

[3] J. Seo and M.S. Lam, "InvisiType: Object-Oriented Security Policies," .In *17th Annual Network and Distributed System Security Symposium*, Internet Society (ISOC), 2010.

[4] A. Yip, X. Wang, N. Zeldovich, and M.F. Kaashoek, " Improving application security with data flow assertions," In *Proc. of the ACM Symposium on Operating Systems Principles*, 2009.

[5] M. Gaimana, R. Simha, and B. Naraharia, "Privacy-preserving programming using sython. Computers & Security, vol. 26, pp. 130, 2007.

[6] B. Cannon and E. Wohlstadter, Controlling access to resources within the python interpreter. Available: http://www.cs.ubc.ca/drifty/ papers/ python security.pdf. 2011.

[7] F.D. Tedesco, A. Russo, and D. Sands, " Implementing erasure policies using taint analysis," *The 15th Nordic Conf. in Secure IT Systems*,Springer Verlag, 2010.

[8] N. Jovanovic, C. Kruegel, and E. Kirda, " Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities,*" In 2006 IEEE Symposium on Security and Privacy,. IEEE Computer Society*, 2006, pp. 258

[9] J.J. Conti and A. Russo, " A taint mode for Python via a library," *NordSec 2010*. Selected paper by OWASP AppSec Research. 2010.

[10] *Goggle Application Engineering*. Available: http://code.google.com/ appengine/ .

[11] A. Petukhov and D. Kozlov, " Detecting security vulnerabilities in web applications using dynamic analysis with penetration testing," *In Proc. of the Application Security Conference*., 2008.

[12] T. Le. Python, *Compiler Internals*. Available: http://www.shinetech.com/attachments/108

[13] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, " BitBlaze: A newapproach to computer security via binary analysis," *In Proc. of the4th International Conference on Information Systems Security*, India, 2008.

[14] Python Package Index. Available: http://pypi.python.org/pypi.

[15] D. Kozlov, A. Petukhov., " Implementation of Tainted Mode approach to finding security vulnerabilities for Python technology.," In *Proc. Of Young Researchers' Colloquium on Software Engineering (SYRCoSE)*, 2007.

[16] O.Tripp, M.Pistoia, S.J.Fink, M.Sridharan, and O.Weisman, "TAJ: effective taint analysis of web applications," In *Proc. ACM SIGPLAN Conference on Programming language Design and Implementation*, ACM Press, 2009.

[17] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood, " Pin: Building CustomizedProgram Analysis Tools with Dynamic Instrumentation," In *Proc. Of the 2005 ACM SIGPLAN conference on Programming language design* and implementation, pages 190. ACM New York, NY, USA, 2005.