Journal of Signal Processing Systems (2022) 94:1169–1182
https://doi.org/10.1007/s11265-022-01740-z

Code Analysis with Static Application Security Testing for Python Program

Li Ma[1] · Huihong Yang[2] · Jianxiong Xu[1] · Zexian Yang[2] · Qidi Lao[2] · Dong Yuan[2]

Received: 29 October 2021 / Revised: 11 December 2021 / Accepted: 30 December 2021 / Published online: 8 March 2022

Abstract
With the increasing popularity of Python for project development, code security and quality have become severe issues for the past few years. The cost of these coding vulnerabilities is hard to estimate and even more costly to fix once the product is released. Besides, the code security audit is inefficient only by manual work, which generally requires tool cooperation. Thus, a Python auditing tool based on *Static Application Security Testing* (SAST) was developed by integrating multiple technologies. Firstly, the tool preprocesses the code to be detected into the *Abstract Syntax Tree* (AST) and performs security analysis by studying the context of the AST and combining it with data flow relationships to determine the existence of vulnerabilities and code security by whether security rules are hit or not. Secondly, to enhance the vulnerabilities detection ability, the tool was designed with plug-in architecture, which allows users to redevelop or rewrite specific rules quickly and easily based on this architecture. Experiments have shown that the SAST technology is fast, efficient, and does not need to configure the environment of code running.

Keywords Code Audit · SAST · Python · Code Security · Vulnerability Detection

1 Introduction

With the development of computer capability [1–3] and algorithm design [4–6], more and more data had been generated by various applications. From January 2019 to January 2021, according to Baidu Index [7] and Google Trends, Python ranked No.1 in China and No.2 in the world respectively, and the search index in China was more than 2.5 times higher than other mainstream programming languages such as Java, C, C# and Go.

Data analysis shows that Python has been developing rapidly in the past two years and has become the preferred language for many projects. However, its low barrier to entry and developers' lack of security awareness [8–10] often led to the lack of security in coding. According to open-source security platform WhiteSource2021 [11], Python accounts for about 5 percent of open-source vulnerabilities in the eight popular programming languages surveyed in 2019-2020, and 6 percent in 2008-2018.

To sum up, today's developed Python applications account for a large base, but the vulnerability mining rate is relatively low, so a *Static Code Audit* (SCA) tool for Python language is designed and implemented.

Code audit [12] is a combination of automated analysis tools and manual review. By analyzing source codes line by line, program errors, security vulnerabilities, and violations of program specifications are found. Code auditing is part of the defensive programming paradigm, which aims to reduce errors and security vulnerabilities caused by these problems before the program is released, and finally to provide code fixes and recommendations. Code audit can be used in various scenarios with the combinations of security approaches in other areas, such as networks [13], smart grid [14], and block chain [15].

This article is part of the Topical Collection on *Big Data Security Track*

✉ Li Ma
molly_917@fosu.edu.cn

Huihong Yang
2112052024@stu.fosu.edu.cn

[1] School of Electronic Information Engineering, Foshan University, Foshan 528000, Guangdong, China

[2] School of Mechatronics Engineering and Automation, Foshan University, Foshan 528000, Guangdong, China

SAST tool helps code auditors improve audit efficiency, which plays a very important role in improving code security. Detection of security problems at the code layer can not only solve product security vulnerabilities from the source but also locate and repair vulnerabilities at a lower cost. Alibaba, Tencent, and Microsoft design the static code audit stage in the SDL (Software Security Development) process in their product security practices or white papers [16–18] , which shows the importance of static code audit tools in code security.

Cobra [19] open-source SAST tool supports vulnerability detection in more than 30 languages such as PHP, Java, and Python, mainly for basic feature scanning, and the introduction of AST is used as an auxiliary means to confirm vulnerabilities, which does not support Windows system. Kunlun mirror [20] open-source SAST tool to achieve semantic analysis by deep rewriting AST, which can improve the effectiveness of vulnerability judgment through multilevel semantic parsing, function backtracking, secret, and other mechanisms. Kunlun Mirror is compatible with Windows and Linux systems but does not support Python. Fortify SCA is a commercial SAST code security audit tool developed by Fortify Software. It supports mainstream languages and provides powerful analysis engines such as semantics, data flow, control flow, and configuration. It also introduces QL [21] technology, which stores the operations of each node in FPR files as state changes. Rule queries can be written in a more general way but it is expensive and has high requirements on the hardware configuration of the runtime environment. Hu Yi [22] extracted semantic features of four code vulnerabilities by combining LSTM, BLSTM, GRU, and GBRU text analysis deep learning methods, to improve the static vulnerability detection accuracy of C/C++. Compared with Checkmarx, Vuddy, and other code auditing tools, the correct rate, accuracy rate and F1 value of the confusion matrix of the research method are as high as 14.2%, 40.5%, and 53.0%. Zhao Wei [23] divided the abstract grammar tree into an expression tree by using the ASTEncoder model through the design of the code vector method and improved the ability of vulnerability location by aggregating expression tree information and combining the attention mechanism. The accuracy rate, recall rate, and F1 value of the model on the test data set reached 98.42%, 96.69%, and 98.03% on average, which were all higher than the traditional method.

The current open-source Python code audit tool is characterized by easy support for the integration of various environments, because the code is open source, but lacks multi-platform compatibility and comprehensive vulnerability detection rules. The mature SAST tool is characterized by supporting most mainstream languages, powerful functions, and providing more detection rules, but the commercial closed source, expensive, cumbersome, and not easy to integrate. In addition, at present, the research objects of SAST related papers are mainly PHP and C/C++ languages, with few studies on Python. The model evaluation mainly has a good effect on the test data sets, without testing the actual projects.

On the basis of previous research, this paper designs a Python code audit tool based on SAST. The unique features of this tool are:

(1) Select the direction of Python code audit, which is less studied at present. According to the characteristics of simple and flat Python development [24], AST is used as the main means to discover the semantics, data flow, and control flow relations of statements, to mine more context information of Python code fragments.
(2) There is no third-party-dependent library native code plug-in architecture design. Plug-ins are Python native code, unlike configuration file plug-ins, which need to abide by specific rules and have various functional limitations, so this tool can be used immediately and can be recognized without reloading meta information. Compared to open-source tools, this tool has 70 built-in detection rules, covering 10 common security problems in Python code security [25].
(3) Compared with the current mainstream SAST code audit tools SonarQube (open source) and Fortify SCA (commercial), this tool has the advantages of higher vulnerability detection rate, lower miss rate, and faster code scanning speed in auditing actual Python open-source projects.

2 Related Work

The SAST in this paper analyzes the code at the Python source level, combining the key information, data flow, and control flow information from the top and bottom of the AST, and analyzes it from different perspectives such as lexical and semantic. By traversal of AST nodes, the key context information is passed to the detection plug-in, and the code is judged to have security problems according to whether the detection plug-in rules are violated.

2.1 Abstract Syntax Tree (AST)

AST is the specific source is expressed as the abstract syntax tree structure, which describes the abstract grammar derivation to the source code statement. In the case of Python, each node of the AST describes a structure that appears in the Python source code and helps us understand what the current syntax looks like at the programming level. In general, the context of the code can be accurately captured through the AST.

2.2 Lexical Analysis

Lexical analysis is the method adopted in the early static analysis, which is the same as the operation in the first step of the compiler. It preprocesses the source code, divides the character stream into words one by one, and matches the word sequence composed of words with the defined unsafe sequence. If the matching is successful, it indicates that the code segment has security holes. G. McGraw [26] judged the vulnerability existence of annotated vulnerability codes in C language through lexical analysis.

2.3 Semantic Analysis

In the field of NLP, semantic analysis [27] uses a variety of methods to identify and understand the semantics of a text so that it can accurately describe a meaning. In particular, an intermediate language can be used to represent its unique meaning. Use the Python language to help you understand the application of semantic analysis in code auditing tools.

2.4 Control Flow Analysis

Control flow analysis [28] relies on a *Control Flow Diagram* (CFG). Where each graph node represents a basic code block, each edge represents a controllable state transition, and the entire CFG+ represents the entire coded control flow diagram. Through the CFG, it can easily handle the context relations of the program. The following is an example of SQL injection in Python.

The code control flow diagram with no SQL injection is shown in Fig. 1. The code control flow diagram with SQL injection is shown in Fig. 2.

It can be found intuitively that there is a user-controllable variable called user in Fig. 2, while there is no user-controllable variable in Fig. 1. According to the context analysis, it is more accurate to judge whether there is vulnerability and the hazard level of the vulnerability.

2.5 Rules of the Test

Rule detection can audit code snippets based on built-in rules or user-defined rules and find code that conforms to the corresponding rules as a result of the detection. Generally, built-in rules are a collection of analysis technologies such as semantics, grammar, control flow, data flow, etc., and rules are written to detect vulnerabilities through the integration of various SAST [29] technologies, such as commercial code auditing tools Fortify SCA and Coverity, which have powerful built-in rules, and

Figure 1 No injection.

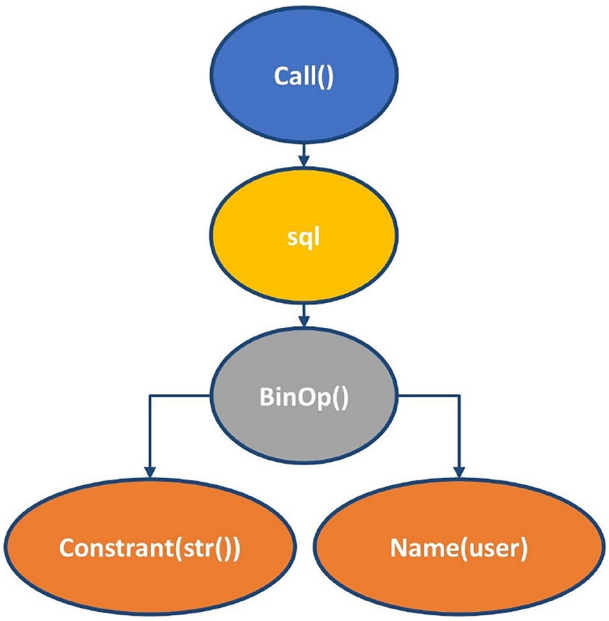

Figure 2 Injection may exist.

open-source auditing tools SonarQube, which can customize rules.

3 Python Code Audit Engine Design

3.1 Architecture Design of Audit Engine

To enhance vulnerability detection and facilitate others to write plug-ins suitable for their scenarios, as well as provide users to fix vulnerabilities, the overall architectural design of this tool is shown in Fig. 3. In order to realize the plug-in architecture of microkernel, this tool mainly includes two modules, a core module and a plug-in module, and six functions, including AST traversal analysis, dynamic plug-in loader, context metadata structure, rule detector, rule plug-in, and report generation plug-in. The plug-in module provides common and easily extensible plug-ins, such as report generation plug-ins for exporting JSON data. In order to ensure the low coupling and maintainability of tool code, the AST traversal analysis, dynamic plug-in loader, rule detector, rule plug-in, and report generation plug-in are implemented separately.

Its overall flow chart is shown in Fig. 4. The engine detection process is, first read the plugin file and the source code to check the sample, the plug-in is loaded into the engine, and then the AST generated from source code, through context access control flow information, such as in static scan through the detector to the source code, according to the code of the plugin detection rules of record, the resulting report, so as to achieve the purpose of automatic code audit.

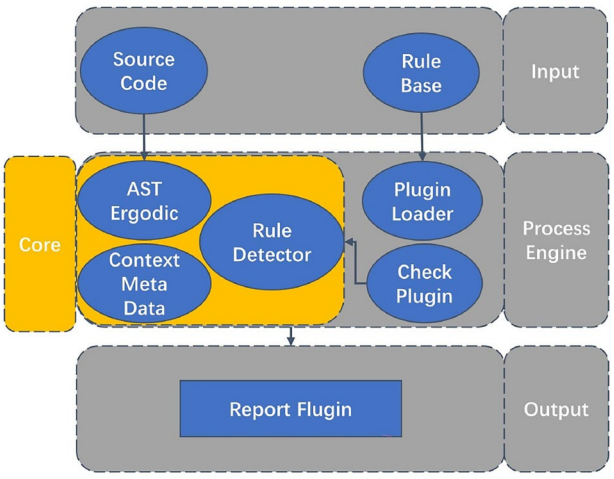

Figure 3 Python code audit engine architecture.

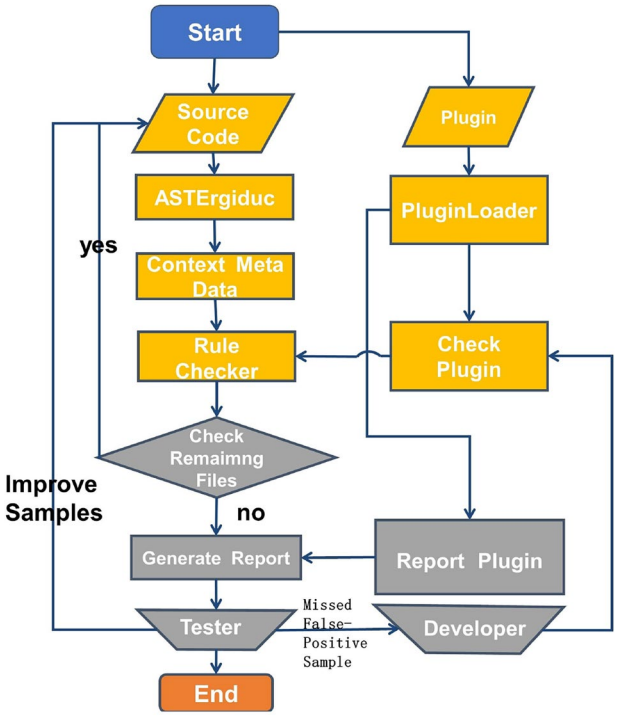

Figure 4 Python code audit engine flowchart.

3.2 Audit Engine Module Design

3.2.1 AST Ergodic Analysis

The AST traversal analysis process is shown in Fig. 5. This step is mainly done by traversing the Python source code AST to get contextual information for each node. When compiling the source code, if the lexical analysis and grammatical analysis fail, the Python parser will throw an exception [30] that can't be parsed properly, so it needs to handle the exception to avoid a program crash.

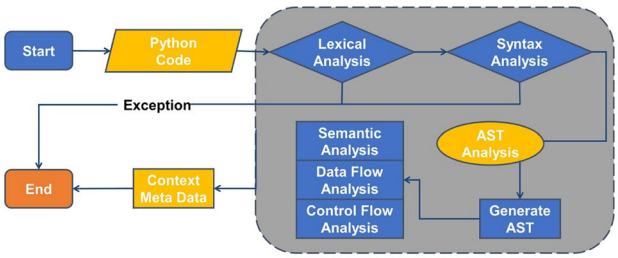

Figure 5 Flow chart of AST ergodic analysis.

3.2.2 Dynamic Plug-in Loader Function

The dynamic plug-in loader function is an important component to realize plug-in. It mainly uses Python's dynamic loading module and reflection mechanism [31, 32] to obtain the object information inside a specific module. It can dynamically load the plug-in information when the program runs.

3.2.3 Rule Plug-in Design

Plug-ins should not only be designed to be easy to use and easy to configure, but also provide rich code context information that can help users achieve the functionality they want to achieve. Here refer to the Python decorator pattern to design the plug-in, decorator to allow other functions to add additional functionality, without any code changes to deal with things, namely, users only need to introduce the necessary decorators that can free writing Python code, according to the engine in the context of the traverse analysis get AST object information to the rules of the design to achieve your requirements, Finally, return the corresponding vulnerability description information according to the triggered user-defined rules. Table 1 begins with a brief list of decorator properties, and Sect. 4.3 describes how to customize the plug-in in detail.

3.2.4 Rule Detector Design

At present, the pollution propagation model [21] is commonly used in SAST technology, which assumes that the data of user input is contaminated data. This view is consistent with the input trust problem [33] – that the user's input cannot be trusted. Look at the code from the point of view that the user input data is a security risk, and judge whether the code has security problems by analyzing the AST context data. Its program flow chart is shown in Fig. 6.

In combination with Fig. 6, it can be found that the interaction between the rule plug-in and the core module is implemented in delegation mode [34] , in which the plug-in plays the role of processor and the core module plays the role of the consigner. If the property is not a callable function, it returns, and if it is, it will be wrapped in a Wrapper as a plugin for the engine's callable object. The advantage of using the delegate pattern is that it simplifies the code and optimizes readability. We don't have to worry about the object being called. The delegate will use getattr to find the method in the corresponding object.

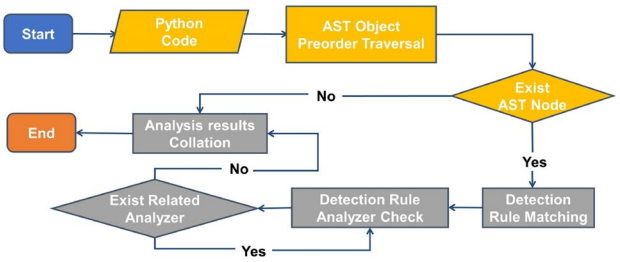

Figure 6 Flow chart of the rule detection function.

3.2.5 Report Plug-in Design

This part is mainly to integrate the information obtained previously and generate information such as vulnerability ID, title, description, severity, confidence degree, location context code of vulnerability, and vulnerability repair information. At the same time, users can refer to the vulnerability repair information in the report to fix the Python code. It is worth noting here that the local file is needed to be opened to obtain the location context code of the vulnerability. If the scanned item is large, there will be some disk I/O overhead, which leads to the slow running speed of the engine. The LazyCache [35] caching mechanism is introduced here, which is a memory-based and thread-safe caching component. The built-in lock does not trigger the cache delegate function when the cache is not hit to reduce the computational overhead.

3.3 Evaluation Indicator Design

In the actual code audit project, the TN condition of the sample confusion matrix (undetected, but potential vulnerability) cannot be accurately known, and the confusion matrix cannot be used for evaluation. Therefore, other indicators need to be defined to compare the detection ability of each tool. Here, the number of detected vulnerabilities, the total number of actual vulnerabilities, the coverage rate, the omission rate, the misjudgment rate, the detection rate, the missing rate and the accuracy rate are respectively defined as:

Table 1 Plug-in decorator description.

Decorator Name	description	options
@pluggable.checks('Call')	Defines a delegate that checks for a function call	required
@pluggable.plugin_id('B201')	Decorator ID	required
@pluggable.takes_config('shell')	Read configuration information under the current plug-in (shell)	optional

```
>>> print(ast.dump(ast.parse('1exec("whoami")'), indent=4))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/Cellar/python@3.9/3.9.2_1/Frameworks/Python.fr
    return compile(source, filename, mode, flags,
  File "<unknown>", line 1
    1exec("whoami")
     ^
SyntaxError: invalid syntax
```

Figure 7 Syntax analysis abnormal lead to program errors.

(1) Number of detected vulnerabilities. Assume that the tool-set participating in the experiment is T, and the number of vulnerabilities in the scanning result of a tool (including the number of misjudged vulnerabilities) is N_i, then the number of vulnerabilities detected is:

$$N_i = TP + FP, i \in T \tag{1}$$

(2) Total number of actual vulnerabilities. The tools detect the union of real vulnerabilities:

$$S_{TP} = \bigcup_{i \subset T} N_i(TP) \tag{2}$$

(3) Coverage rate. Set the total number of vulnerabilities in test samples as S_{test}, then the coverage rate represents the ratio of the true number of vulnerabilities in the scanning results of a tool to the total number of vulnerabilities in test samples:

$$P_{Ci} = \frac{N_i(\mathrm{TP})}{S_{\mathrm{test}}}, i \in T \tag{3}$$

(4) Omission rate. The omission rate for a tool is:

$$P_{\bar{C}i} = 1 - P_{Ci}, i \in T \tag{4}$$

Table 2 AST object description

ID	Ast object name	description
1	Import,ImportFrom	module import
2	Name	variable name
3	Call	function call
4	ClassDef	class definition
5	FunctionDef	function definition
6	Constant	constant
7	Str	string object
8	Assign	assignment statement
9	Assert	assert statement
10	ExceptHandle	exception handling node
11	BinOp	binary operator

```
>>> print(ast.dump(ast.parse('exec("whoami")'), indent=4))
Module(
   body=[
      Expr(
         value=Call(
            func=Name(id='exec', ctx=Load()),
            args=[
               Constant(value='whoami')],
            keywords=[]))],
   type_ignores=[])
```

Figure 8 Generate AST through grammar and lexical analysis.

(5) Misjudgment rate. The ratio of the number of misjudged vulnerabilities of a tool to the number of detected vulnerabilities of a corresponding tool:

$$P_{Fi} = \frac{N_i(FP)}{N_i}, i \in T \tag{5}$$

(6) Detection rate. The ratio of the number of vulnerabilities detected by one tool and the number of sets (including the number of false positives) detected by the participating experimental tools:

$$P_{Ti} = \frac{N_i}{U_{i \in T} N_i}, i \in T \tag{6}$$

(7) Missing rate. The ratio of the difference between the total number of true vulnerabilities and the total number of true vulnerabilities of a tool

$$P_{Mi} = 1 - \frac{N_i(TP)}{S_{TP}}, i \in T \tag{7}$$

(8) Accuracy rate. The proportion of the true number of vulnerabilities of a tool is corresponding to the ratio of the number of detected vulnerabilities of a tool:

$$P_{Ai} = \frac{N_i(TP)}{N_i}, i \in T \tag{8}$$

The true number of vulnerabilities of accuracy is determined according to the context and parameter stream of manual analysis code, so there may be some deviation from the actual results. The higher the coverage, the better the tool can cover existing or potential security issues in the code. The higher the accuracy, the better the tool can mine the real exploitable vulnerabilities; The higher the detection rate, the more vulnerabilities can be detected in the comparison tool; The lower the missed detection rate, the stronger the mining ability of a tool in the contrast detection tool.

4 Python Code Audit Engine Implementation

4.1 Implementation of AST Ergodic Analysis

The lexical and syntactic analysis of Python source code files can be done using Python's standard library AST, which generates the AST after the lexical and syntactic analysis are done. If Python scripts are not normal execution of analysis of failure, as shown in Fig. 7, try to use Numbers at the beginning of the function name variable names, throw grammar error exception.

The Python3 official documentation lists all the AST node objects. Table 2 provides a brief description of the main AST object types used by the engine.

By combining Fig. 8 and Table 2, we can see that Python's abstract syntax tree is an N-tree with AST objects as nodes, as shown in Fig. 9.

Each node can be obtained through the recursive preorder traversal algorithm of the tree, and the implementation algorithm is shown in Algorithm 1. Therefore, we can know that the time complexity of Algorithm 1 is O(n).

Algorithm 1 Abstract syntax tree traversal analysis

Input: ast_root $< ast.Module >$
Output: Context information for each node $< datatype.Context >$

```
function node_visit(node : ast.AST)
    for all ast_node ∈ node → ast_root do
        if isinstance(ast_node, list) then
            i_max = len(ast_node) − 1
            for i = 0 to len(ast_node) do
                if not isinstance(ast_node(i), ast.AST) then
                    return
                end if
                if i < i_max then
                    // for find the location of next line
                    ast_node(i).sibling = ast_node(i + 1)
                else
                    ast_node(i). = None
                end if
                Call visit(ast_node(i))
                Call node_visit(ast_node(i))
            end for
        else if isinstance(ast_node, ast.AST) then
            ast_node(i). = None
            Call visit(ast_node(i))
            Call node_visit(ast_node(i))
        end if
    end for
end function
```

For the traversed nodes, we need to get the context and deep information (semantics, control flow, etc.) of the current node, that is, the Call visit calling function in Algorithm 1. Here, the context, semantics and control flow information of ast.Call and ast.Import are obtained by taking ast.Call and ast.Import objects as an example, which is implemented as shown in Algorithm 2. Since the algorithm uses the function signature as the key to obtain the function run address value stored in the Python runtime context hash table, which is Python's reflection dynamic call function mechanism, the time complexity of the algorithm is O(1).

4.2 Implementation of Dynamic Plug-in Loading

Python provides high-level functions for the introduction of certain namespace modules *__import__(name, globals=None, locals=None, fromlist=(), level=0)*, This function imports the named module and focuses on the formalist argument for normal calls because by default this function will only return the highest-level package (the name up to the first dot). For example, *__import__('plugins.exec')* only return package plugins. If you want to import the plugins. exec module, just make sure the formalist parameter is not empty, such as *__import__('plugins.exec', fromlist=[':')]).*

Algorithm 2 Save the node context information

Input: $node < ast.AST >$

Output: Save the current node context information $< datatype.Context >$

```
1:  function visit(node : ast.AST)
2:      context → datatype.Context //context data
3:      ast_node(i).parent = node
4:      for all context.property ∈ current namespace do
5:          context.property → current namespace property
6:      end for
7:      // Drill down to the AST object information
8:      name → node.class.name___
9:      method →' visit_' + name
10:     visitor = getattr(method, None)
11:     if visitor ≠ None then
12:         Call visitor(node)
13:     end if
14: end function
15:
16: function visit_Import(node : ast.AST)
17:     for all node_name ∈ node.names do
18:         if node_name.asname then
19:             aliases[node_name.asname] = node_name.name
20:         end if
21:     end for
22: end function
23:
24: function visit_Call(node : ast.AST, aliases)
25:     if isinstance(node.func, ast.Name) then
26:         if getattr(node.func,' id') ∈ aliases then
27:             return aliases[getattr(node.func,' id')]
28:         end if
29:         return getattr(node.func,' id')
30:     else if isinstance(node.func, ast.Attribute) then
31:         return _get_attr_qulify_name(node.func, aliases)
32:     else
33:         return str()
34:     end if
35: end function
```

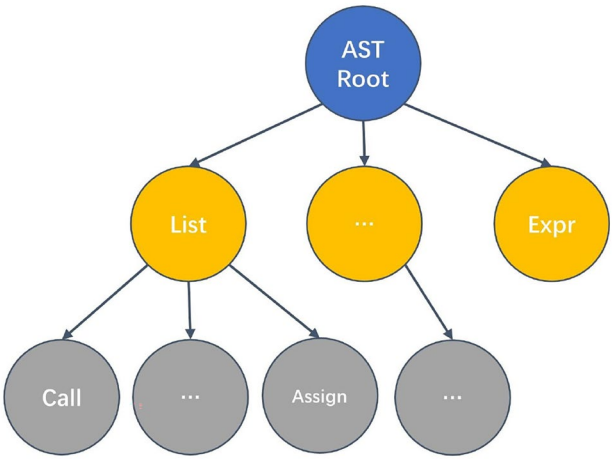

Figure 9 AST multi-tree structure.

Python's reflection mechanism is implemented using the getattr() function. The detection rule plug-in of this engine is divided into custom rule and blacklist rule. The entry function is identified by the function name prefix of the plugin namespace, such as the entry function name starts with a process and the blacklist plug-in starts with gen_. For importing module functions, getattr(Imported, Func) can be used to obtain the internal function information of a particular module, and the plug-in information can be loaded dynamically when the program is running.

4.3 Implementation of Rule Plug-in

In addition to the prefix of the plugin function name mentioned in Sect. 4.2, we also need to set the plugin decorator mentioned in Sect. 3.2.3, which sets the plugin meta information. Figure 10 illustrates the example of the danger detection function exec plugin.

For each plug-in, you must start with Lines 1-6 of the plug-in's core dependency modules, including:

```
@pluggable.checks('Call')
@pluggable.plugin_id('B102')
def process(context: datatype.Context):
    if context.func_qualify_name == "exec":
        if len(context.call_args) > 0:
            if isinstance(context.node.args[0], ast.Str):
                return issue.Issue(
                    severity=enums.VulnRank.MEDIUM,
                    confidence=enums.VulnRank.HIGH,
                    text='检测到 exec 函数使用'
                )
            else:
                return issue.Issue(
                    severity=enums.VulnRank.HIGH,
                    confidence=enums.VulnRank.HIGH,
                    text='检测到参数可控 exec 函数使用'
                )
```

```
from core import (
    enums,
    issue,
    datatype,
    pluggable
)
import ast
```

Figure 10 Dangerous function exec detection plug-in.

(1) Enum is enumeration type, which contains common constants, such as vulnerability hazard level, the rule's judgment confidence, and so on.
(2) Issue is used to collect information such as vulnerability description and vulnerability level of this rule, which is used for information integration by report generation plug-in.issue.
(3) Datatype contains the AST context information data structure and other data structure.
(4) Pluggable is a plug-in decorator. In addition to the process prefix, the custom rule plug-in is needed to specify the ID of the plug-in and the plug-in check type. The type referred as pluggable can check the rules when it detects a node of the AST type, such as Call function call and Assign value. If the information is incomplete, the core module will not load the plug-in and send an alert message. There's also an optional decorator *@plugable.takes_config('XXX')* that reads the *gen_config(name)* function of the plugins file for the plugins XXX, which can correspond to the name of *gen_config* which represents the detection plug-in entry function that calls the configuration function.

The detection rule is the integration process of the context information extracted from the AST and the vulnerability characteristics. That is, for the exec function, the AST will

Figure 11 Report of code audit results.

```
from core.issue import Issue
from core.controller.manager import Manager


def report(manager: Manager, output_file, severity_level, confidence_level, lines=-1):
    """
    The scanning result is output to the terminal,
    and the VT100 terminal control code is used to output the color

    :param manager: core.controller.manager.Manager object
    :param output_file: Output file path, terminal's output_file.name is <stdout>
    :param severity_level: Outputs vulnerabilities greater than this severity
    :param confidence_level: Outputs vulnerabilities greater than this confidence
    :param lines: Output the number of lines of vulnerability location, - 1 shows the entire file content
    """
    pass
```

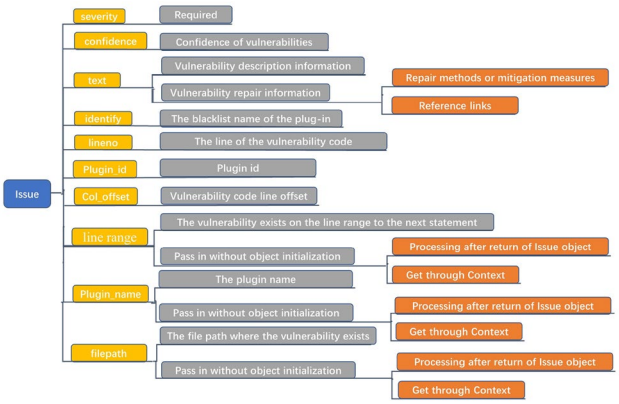

Figure 12 Issue vulnerability information field description.

have a Call object. According to the semantic information obtained in Sect. 4.1, in Line 13 of Fig. 10 if the semantic name of the called function is exec, it means that exec will be called to check if any parameters are present in the follow-up context data flow information as showed in line 14. The command execution function should not contain controllable parameters which make it easy to be used by attackers. That is, in line 15 of code, if the parameters are controllable (the ast.Str instance is a constant string object), it is judged that there may be a high-risk vulnerability in the code; otherwise, it is judged that there may be a medium-risk vulnerability, because the function may be modified in the future which causes being intentioned.

4.4 Implementation of Regular Detector

The rule detector is located in the core module and interacts with the detection rule plug-in through the delegate mode. The rule detector reads all plug-ins after the initialization of the rule plug-in and can be directly called by the delegate for subsequent use.

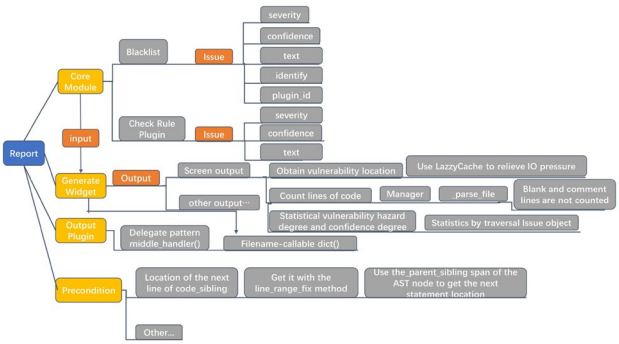

Figure 13 Relationship between the report generation plug-in and the core module.

Table 3 Test sample description.

Name of vulnerability	Line Number	N_{TP}	N_{FP}
SQL Injection	124	30	19
XSS	243	66	12
XXE	144	35	9
Command Injection	254	64	42
SSTI	27	5	4
Insecure Deserialization	104	15	4
Information Leakage	107	14	5
Cryptographic Security Issues	306	80	20
Unclear Access Control Boundaries	134	40	12
Insecure Server-Side Configuration	106	20	11
Statement Logic Defects	74	7	2

During detection, the detector calls corresponding rule detection through the client itself according to the AST object type (Call, Import, etc.). The checker is the handler, so we don't have to care about the specific object to be called.

4.5 Implementation of Report Plug-in

The core modules needed by this plug-in are illustrated by the example of the terminal report generation plug-in in Fig. 11 below.

The output plug-in is identified as the entry function with the prefix "report". There are 5 input parameters in this function, and the last 3 parameters can be described in Fig. 11. As for the second parameter, it is usually the path of the output file, so the plug-in is used to output to the screen. The file descriptor $< stdout >$ is usually returned to indicate terminal output.$< stdout >$. As for the first parameter Manager, it is the management object of the core component, which mainly contains the measurement of the scan result (vulnerability hazard level, confidence degree, number of vulnerabilities), the number of lines of code scanned, and the issues of vulnerability information collected in the rule plug-in.

The list of vulnerability information collectors can obtain through the method in Code Snippet 1. Each issue contains vulnerability information, as shown in Fig. 12. The logical relationship between the report generating plug-in and the core module can be seen in Fig. 13.

Table 4 Each tool test results contrast test sample.

Tool	P_{Ci}	$P_{\tilde{C}i}$	P_{Fi}	N_i
PSCAT	80.07%	19.93%	35.00%	320
SonarQube	15.29%	84.71%	5.35%	56
Fortify SCA	16.49%	83.51%	3.22%	64

Code Snippet 1 :

```
issues: list[Issue] = manager.get_issue_list(
    severity_level, confidence_level
)
```

5 Experiment with Python Auditing Tools

5.1 Introduction to the Experimental Environment

PSCAT (Python Static Code Analysis Tool, this name is hereafter used to represent the name of the tool) is a SAST tool developed based on Python3 and compatible with Windows, Linux and MacOS platforms. It can be compatible with the Python3.6+ environment through CI/CD Build test. There are 70 built-in detection rules. It covers 11 common Python code security vulnerabilities, including SQL Injection, XSS, XXE, Command Injection, SSTI, Insecure Deserialization, Information Leakage, Cryptographic Security Issues, Unclear Access Control Boundaries, Insecure Server-Side Configuration, and Statement Logic Defects.

This experiment only compares the static code audit results of PSCAT and other tools in Python language. The experimental control group is SonarQube(an open source static code audit tool) and Fortify SCA(a commercial static code audit tool). Both of them are implemented based on SAST technology, so it is of comparative significance. The number of lines of scanned code, number of vulnerabilities, scanning time and other data were counted according to the operation logs and results of each tool.

PSCAT running environment: Python3.6+, hardware configuration of 2-core CPU and 4G operating memory.

SonarQube CE 8.8 running environment: Ubuntu20.04 Server, with Docker deployment of GitLab, Jenkins, and SonarQube CI/CD environments, the hardware configuration of 4 cores CPU and 16GB of running memory.

Fortify SCA 20.1.1.0007 running environment: Windows10 LTSC, installed through the official website installation package, hardware configuration of 8-core CPU and 24GB running memory

5.2 Test Sample Experiment

The test sample is the test code that covers common Python vulnerabilities, and the information is shown in Table 3.

Table 3 covers 11 common vulnerabilities with 1623 lines of Python test code, which were scanned using PSCAT, SonarQube, and Fortify SCA tools, respectively. The test results are shown in Table 4 and Bar Fig. 14.

From the test sample results, the PSCAT has a higher misjudgment rate of 31.78% compared with the other two tools, but the coverage rate is 63.58%, which can better detect the security vulnerabilities in the samples.

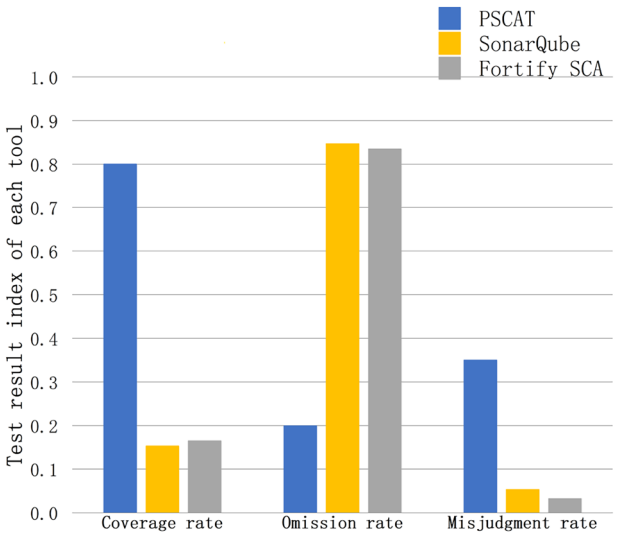

Figure 14 Each tool test results contrast test sample.

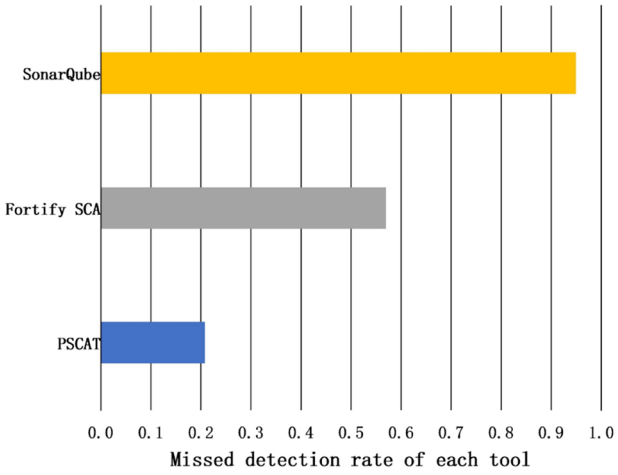

Figure 15 AST multi-tree structure Comparison of missed detection rate of each tool.

Table 5 Open Source Sample Description.

ID	Sample Item Functions	Sample lines of code	GitHub Star number
1	Instant chat software automation framework	4737	336
2	SRC asset management system	10490	945
3	Campus educational administration system interface SDK	1745	165
4	Questionnaire survey system	796	52
5	Simple diagram management system	333	72
6	IT asset management system	4386	59
7	Automatic operation and maintenance system	3293	212
8	The Abu quantitative trading system	41759	7.8k
9	Blog system	5763	3.7k
10	Stock system	2107	4.2k
11	Blog system	9527	271
12	Automated operation and maintenance management system	7224	227
13	Web monitoring system	23574	71
14	Security audit system	35766	15.7k

5.3 The Open-source Project Sample Experiment

The main scenario of PSCAT is the code audit of Web applications and applications, so the first step is to find Python open-source projects related to Web applications and application systems through GitHub search syntax.

The selection of open-source samples should include different aspects of the application, different sizes of code, and different users. Sample information of open-source projects is shown in Table 5.

Different open-source application systems, such as security audit, operation and maintenance, blog, asset management, transaction, chat, and questionnaire, were sampled successively. A total of 151,500 lines of Python code were sampled and scanned using PSCAT, SonarQube, and Fortify SCA tools, respectively. The experimental results are shown in Tables 6 and 7 respectively.

The bar chart of missed detection rate of each tool is shown in Fig. 15.

Each tool that scans 151500 lines of code needed to consume the relation between time and space is shown in Table 8.

Tables 6, 7, 8 and Figs. 14, 15 show that in the 151500 lines of code of this open source sample, the detection rate of PSCAT is 21%+ higher, the miss rate is 36.21%+ lower and the scanning speed is 528.06+ lines/(kernel*second) faster than that of SonarQube and Fortify SCA, and consumed time 15.2 seconds faster. The accuracy of PSCAT was 63.26% lower than that of SonarQube, and the operating memory cost was 29M higher than that of SonarQube. PSCAT has certain advantages over Fortify SCA in various evaluation indicators, and has certain advantages over SonarQube in detection rate, missed detection rate and scanning speed. The experimental results show that PSCAT can be used in Python code audit of actual open-source projects on the condition that it is compatible with multiple platforms, with lower time and space complexity of program running and certain accuracy.

Table 6 Open-source samples each tool test results contrast.

Tool	P_{Ti}	P_{Ai}	P_{Mi}
PSCAT	71.27%	34.76%	20.69%
SonarQube	1.62%	100%	94.85%
Fortify SCA	50.27%	26.79%	56.90%

Table 7 Open-source samples each tool test results contrast.

Tool	N_i	running time
PSCAT	397	82.85s
SonarQube	9	98.06s
Fortify SCA	280	13225.42s

Table 8 Time and space complexity of each tool.

Tool	*Scanning speed*(*lines*/(*cores* ∗ *seconds*))	Run average memory usage
PSCAT	914.30	66MB
SonarQube	386.24	37MB
Fortify SCA	1.43	$\geqslant 8GB$

6 Summary

This paper chooses popular programming languages with a low proportion of open source vulnerabilities as the research target. The experimental results show to some extent that there is no such thing as a "safe" language, and more of it is that there is not enough security work in the whole development life cycle. SAST code audit tool is an important technology to detect software security vulnerabilities. It can detect potential security problems in the code and solve security problems more easily before the code is officially published. Code audit work is usually completed by professionals such as penetration testers, SDL engineers, code audit engineers, etc. It usually takes a high cost of manpower and time, and the quality of the audit is related to the working life of the engineer. For complex projects, the level of human brain limitations will also be highlighted. For small companies and individual developers, this cost is also unaffordable. Therefore, the SAST code audit tool is very necessary, It can cover the code simpler and more comprehensively, and examine the security question of the code from the underlying perspectives of the code's lexical, grammatical, semantic, data flow, control flow, and code structure levels. By taking the pre-generated AST and the code to be detected as the research object, this project analyzes the context relationship of the AST, designs PSCAT in a plug-in way, writes more than 70 detection rule plug-ins covering common vulnerabilities, determines the existence of code vulnerabilities in the form of hitting security rules, and finally generates intuitive reports. PSCAT has the advantages of cross-platform, low runtime and space complexity, high coverage, etc. It has the functions that the SAST tool should have, and can be integrated into the current mainstream software development process, such as SDL and DevSecOps, in this way to engineering PSCAT.

There are still some areas that can be improved in PSCAT, such as the false positive rate needs to be reduced and the code detection rule data needs to be increased. In the future, PSCAT can also integrate the concept of QL just like CodeQL, that is, it can visualize the semantics, data flow, and control flow obtained through the AST context relationship, and store the operation of each node in the database as the state change. This allows us to write rule queries in a more generic way (unlike the PSCAT rule detection plug-in, which is Python only) by moving a requirement to a higher level and simplifying the query to satisfy the characteristics of the vulnerability.

Funding This work was supported by grants from the Natural Science Foundation of Guangdong Province No.2018A0303130082, The Features Innovation Program of the Department of Education of Foshan No.2019QKL01, Basic and Applied Basic Research Fund of Guangdong Province No.2019A1515111080, and Natural Science Foundation of China No.61802061.

References

1. Qiu, M., Chen, Z., Niu, J., Zong, Z., Quan, G., Qin, X., & Yang, L. T. (2015). Data allocation for hybrid memory with genetic algorithm. *IEEE Transactions on Emerging Topics in Computing, 3*(4), 544–555.
2. Qiu, M., Sha, E. H. -M., Liu, M., Lin, M., Hua, S., & Yang, L. T. (2008). Energy minimization with loop fusion and multi-functional-unit scheduling for multidimensional dsp. *Journal of Parallel and Distributed Computing, 68*(4), 443–455.
3. Zhao, H., Chen, M., Qiu, M., Gai, K., & Liu, M. (2016). A novel pre-cache schema for high performance android system. *Future Generation Computer Systems, 56,* 766–772.
4. Qiu, M., Ming, Z., Li, J., Liu, J., Quan, G., & Zhu, Y. (2013). Informer homed routing fault tolerance mechanism for wireless sensor networks. *Journal of Systems Architecture, 59*(4–5), 260–270.
5. Li, J., Qiu, M., Niu, J., Gao, W., Zong, Z., & Qin, X. (2010). Feedback dynamic algorithms for preemptable job scheduling in cloud systems. In *IEEE/WIC/ACM Conference on Web Intelligence*.
6. Tang, X., Li, K., Qiu, M., & Sha, E. H. -M. (2012). A hierarchical reliability-driven scheduling algorithm in grid systems. *Journal of Parallel and Distributed Computing, 72*(4), 525–535.
7. Baidu. (2021). *Baiduindex*. https://zhishu.baidu.com/v2/main/index.html#/trend/python?words=python,java,c. Accessed 25 April 2021.
8. Gai, K., Qiu, M., Thuraisingham, B., & Tao, L. (2015). Proactive attribute-based secure data schema for mobile cloud in financial industry. In *IEEE 17th HPCC*.
9. Thakur, K., Qiu, M., Gai, K., & Ali, M. L. An investigation on cyber security threats and security models. In *CSCloud'15* (pp. 307–311).
10. Gai, K., Qiu, M., Sun, X., & Zhao, H. (2016). Security and privacy issues: A survey on FinTech. In *SmartCom* (pp. 236–247).
11. WhiteSource. (2021). *All about whitesource's 2021 open source security vulnerabilities report*. https://www.whitesourcesoftware.com/resources/blog/2021-state-of-open-source-security-vulnerabilities-cheat-sheet/. Accessed 25 April 2021.
12. Thomé, J., Shar, L. K., Bianculli, D., & Briand, L. (2018). Security slicing for auditing common injection vulnerabilities. *Journal of Systems and Software, 137,* 766–783.
13. Zhang, Z., Wu, J., Deng, J., & Qiu, M. (2008). Jamming ACK attack to wireless networks and a mitigation approach. In *IEEE GLOBECOM* (pp. 1–5).
14. Su, H., Qiu, M., & Wang, H. (2012). Secure wireless communication system for smart grid with rechargeable electric vehicles. *IEEE Communications Magazine, 50*(8), 62–68.
15. Qiu, H., Qiu, M., Memmi, G., Ming, Z., & Liu, M. (2018). A dynamic scalable blockchain based communication architecture for IoT. In *Int'l Conf. on Smart Blockchain* (pp. 159–166).
16. Alibaba Group. (2018). *Dingtalk security white paper*. http://download.taobaocdn.com/freedom/33465/pdf/dingtalksecuritywhitepaperV1.0.pdf. Accessed 11 May 2021.
17. Tencent Cloud. (2019). *Tencent cloud security white paper*. https://main.qcloudimg.com/raw/cf70533c2a60ec66e5c133cc0a02a92e.pdf. Accessed on 11 May 2021.
18. Microsoft. (2021). *Microsoft security development lifecycle practices*. https://main.qcloudimg.com/raw/cf70533c2a60ec66e5c133cc0a02a92e.pdf. Accessed 11 May 2021.
19. WhaleShark-Team. (2017). *Cobra*. https://cobra.feei.cn/. Accessed 20 April 2021.
20. LoRexxar. (2020). *Talking about automated static code auditing tools from scratch*. https://paper.seebug.org/1339/. Accessed 11 May 2021.
21. Rodriguez-Prieto, O., Mycroft, A., & Ortin, F. (2020). An efficient and scalable platform for java source code analysis using overlaid graph representations. *IEEE Access, 8,* 72239–72260.

22. Hu, Y. (2020). *An approach for using deep learning to detect code vulnerabilities*. PhD thesis, Harbin Institute of Technology.
23. Zhao, W. (2020). *An approach for using deep learning to detect code vulnerabilities*. PhD thesis, Harbin Institute of Technology.
24. Walden, J. (2020). The impact of a major security event on an open source project: The case of openssl. In *Proceedings of the 17th International Conference on Mining Software Repositories* (pp. 409–419).
25. veracode. (2021). *Security flaw heat map*. https://www.veracode.com/sites/default/files/pdf/resources/ipapers/security-flaw-heatmap/index.html. Accessed 19 April 2021.
26. McGraw, G. (2008). Automated code review tools for security. *Computer, 41*(12), 108–111.
27. Kim, S., Park, H., & Lee, J. (2020). Word2vec-based latent semantic analysis (w2v-lsa) for topic modeling: A study on blockchain technology trend analysis. *Expert Systems with Applications, 152,* 113401.
28. Burow, N., Carr, S. A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., & Payer, M. (2017). Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR), 50*(1), 1–33.
29. MoreSec. (2019). *A text insight DAST, SAST, IAST - web application security testing technology comparison talk*. https://www.aqniu.com/learn/46910.html. Accessed 11 April 2021.
30. Yuanying, X., Yaodong, Y., & Lixi, X. Causes and optimization of the false alarm rate of code review system. *Telecommunications Science, 36*(12), 155.
31. Lu, Q., Jiang, R., Ouyang, Y., Qu, H., & Zhang, J. (2020). Bire: A client-side bi-directional SYN reflection mechanism against multi-model evil twin attacks. *Computers & Security, 88,* 101618.
32. Nielson, J. A. (2018). A Jython-based RESTful web service API for python code reflection.
33. McGraw, G., & Hoglund, G. (2004). Exploiting software: How to break code. In *Invited Talk, Usenix Security Symposium, San Diego* (pp. 9–13).
34. Python Practical Dictionary. (2020). *Understanding design patterns with python - delegation pattern*. https://pythondict.com/python-solution/python-understand-delegate-pattern/. Accessed 20 April 2021.
35. Banks, C. J., Elver, M., Hoffmann, R., Sarkar, S., Jackson, P., & Nagarajan, V. (2017). Verification of a lazy cache coherence protocol against a weak memory model. In *2017 Formal Methods in Computer Aided Design (FMCAD)* (pp. 60–67). IEEE.