Computers, Materials & Continua
DOI: 10.32604/cmc.2023.034505
Article

Tech Science Press

JShellDetector: A Java Fileless Webshell Detector Based on Program Analysis

Xuyan Song, Yiting Qin, Xinyao Liu, Baojiang Cui* and Junsong Fu

School of Cyber Security, Beijing University of Posts and Telecommunications, Beijing, 100876, China
*Corresponding Author: Baojiang Cui. Email: cuibj@bupt.edu.cn
Received: 19 July 2022; Accepted: 22 September 2022

Abstract: Fileless webshell attacks against Java web applications have become more frequent in recent years as Java has gained market share. Webshell is a malicious script that can remotely execute commands and invade servers. It is widely used in attacks against web applications. In contrast to traditional file-based webshells, fileless webshells leave no traces on the hard drive, which means they are invisible to most antivirus software. To make matters worse, although there are some studies on fileless webshells, almost all of them are aimed at web applications developed in the PHP language. The complex mechanism of Java makes researchers face more challenges. To mitigate this attack, this paper proposes JShellDetector, a fileless webshell detector for Java web applications based on program analysis. JShellDetector uses method probes to capture dynamic characteristics of web applications in the Java Virtual Machine (JVM). When a suspicious class tries to call a specific sensitive method, JShellDetector catches it and converts it from the JVM to a bytecode file. Then, JShellDetector builds a Jimple-based control flow graph and processes it using taint analysis techniques. A suspicious class is considered malicious if there is a valid path from *sources* to *sinks*. To demonstrate the effectiveness of the proposed approach, we manually collect 35 test cases (all open source on GitHub) and test JShellDetector and only two other Java fileless webshell detection tools. The experimental results show that the detection rate of JShellDetector reaches 77.1%, which is about 11% higher than the other two tools.

Keywords: Web security; fileless webshell; Java web application; malware

1 Introduction

In the past few years, injection vulnerabilities have been in the Open Web Application Security Project (OWASP) Top 10 [1], which makes web servers the largest victim of network attacks. Attackers use injection vulnerabilities to upload malicious programs and scripts to web servers or inject existing files to damage websites [2]. A webshell is a typical malicious script that exploits an injection vulnerability to launch a persistent attack. Webshell is a shell-like interface that can remotely execute commands and hack servers. Once the server is implanted with a webshell, an attacker can legitimately connect to the server by loading the webshell page in the browser. Through the webshell page, attackers

can steal information, tamper with databases, upload more dangerous malware and escalate their privileges [3]. The problem is exacerbated when attackers use the compromised server as a springboard to break into internal network systems.

A fileless webshell is a special kind of webshell that is sometimes referred to as fileless malware or fileless attack. Compared to a traditional webshell, it only runs in memory (system memory or virtual machine memory), does not rely on files, or deletes its fingerprints after running, which means it is an invisible attack. This feature makes both detection and removal a challenge. According to the 2020 fourth quarter security report released by the network security research company WatchGuard [4], the infection rate of fileless malware increased by 888% in 2020 compared with 2019. In the same year, cybersecurity research company Red Canary published a threat detection report [5], detailing that 48.7% of organizations were affected by malicious fileless webshells.

Because the fileless webshell deletes itself and stays hidden in memory, it has high concealment and is extremely difficult to detect. Therefore, how to quickly and accurately detect whether a fileless webshell is implanted in a server is crucial to the security of systems. In recent years, there have been many studies in this field. Webshell detection is usually divided into static detection and dynamic detection. Static detection does not require the execution of webshell but models malicious behavior through statistical features. For example, Kang et al. [6] used statistical features such as information entropy and coincidence index of PHP source files to extract opcode sequences of PHP source files, and integrated statistical features with opcode sequences to improve the detection efficiency of webshell. Yang et al. [7] realize the detection of webshell attack traffic in network traffic using the method of supervised machine learning model. Compared with static detection, dynamic detection takes more time and may produce a large number of false positives [8]. Therefore, the mainstream method for webshell is static detection.

Through literature research, we found that the existing research on webshell detection has two characteristics. On the one hand, most of the research is for traditional file-based webshells, and research on fileless attacks focuses on the Windows operating system rather than web servers. On the other hand, PHP webshell detection has attracted the attention of most researchers. Java as one of the most widely used program languages in the world [9], has not received corresponding attention. To the best of our knowledge, there is currently no literature for fileless webshell detection in Java web applications. We consider that this dilemma stems from three challenges: 1) Lack of samples of Java fileless webshells. There is no publicly available dataset on the internet, sporadic webshells samples exist in blogs. 2) A variety of Java web frameworks. Most Java web applications are built with open source frameworks that encapsulate many low-level operations. These complex features provide convenience to attackers to implant fileless webshell while increasing the difficulty of detection. 3) Java Virtual Machine (JVM). The JVM is an independent ecosystem, which makes the fileless malware detection method for PHP applications or the Windows operating system unable to be directly migrated to Java web applications.

In this paper, we overcome the above 3 challenges and propose JShellDetector, a detector for Java fileless webshells based on JVM monitoring and taint analysis. Overall, the main contributions of this paper are as follows:

1) This paper studies the principle of Java fileless webshell in detail, and constructs a webshell test case dataset based on different Java web components and Spring framework components. All test cases are publicly accessible.
2) We propose JShellDetector, a Java fileless webshell detector based on Java Virtual Machine monitoring and taint analysis. The method probe proposed in this paper can be triggered when

a suspicious class calls a sensitive method and converts the class into a bytecode file. Then taint analysis is used to determine whether the suspicious class is a webshell.

3) We evaluate the performance of our tools in detail and compare them with the only two Java fileless webshell detection tools. The experimental results show that JShellDetector can find more webshell test cases. The detection rate of our tool is 77.1%, which is about 11% higher than the other two tools.

The remainder of this paper is organized as follows. In Section 2, we introduce the related work about fileless webshell detection. Two mainstream approaches are discussed in detail. Section 3 introduces the background knowledge of Java fileless webshell, including the composition of Java web and the classification of Java fileless webshell. In Section 4, the composition and principles of JShellDetector are introduced in detail. To evaluate JShellDetector, we make comprehensive experiments in Section 5. We propose 3 standard research questions and demonstrate the effectiveness of our tool by answering them. Section 6 summarizes the work in this paper and introduces future work.

2 Related Work

Webshell detection is usually divided into static detection and dynamic detection. In this section, the classification models and advantages and disadvantages of these two detection approaches are introduced respectively.

2.1 Static Webshell Detection

Static detection finds webshells by matching statistical features and code features. The advantage of this approach is that it is easy to deploy, with a simple script. At the same time, static detection has a high detection rate for known webshell features. But this method can only find known webshells and requires manual exclusion of weak signature files. This process requires a lot of manual work.

Zhu et al. [10] proposed a webshell detection system based on the support vector machine (SVM) classifier. This classifier is trained and tested on mixed source code features, i.e., lexical, syntactic and statistical. A total of 20 features were first measured and normalized, then filtered using the Fisher score to designate the most important features. Finally, 16 features were selected to build the SVM model.

Kurniawan et al. [11] used static statistical features based on source code. They define 50 sensitive functions and count the number of calls to these functions in each file. The obtained counts are used to construct feature vectors, which are used to provide different classifiers. Experiments show that SVM performs better than other classifiers such as Decision Tree and Naive Bayes. However, the dataset used in the experiments is not limited to 100 files, so more data is needed to validate the model.

Huang et al. [12] used 2 statistical features, 21 high-risk functions, and term frequency-inverse document frequency (TF-IDF) vectorization of PHP opcode sequences. They found that the random forest (RF) classifier outperformed the SVM and k-nearest neighbors (KNN) algorithms. The advantage of this approach is that it combines two opcode sequences, one for the execution path and one for the code itself. Experimental results show that the proposed model shows improved performance.

Tian et al. [13] proposed a word2vec-based feature extraction method. First, they use the word2vec tool to convert each word of the Hypertext Transfer Protocol (HTTP) request into a vector. Then the scheme converts the vector to a fixed-size matrix. Finally, detection methods based on convolutional neural network (CNN) models are used for classification.

2.2 Dynamic Webshell Detection

Dynamic detection determines the threat level of the behavior through the system commands invoked by the webshell runtime. Therefore, it is necessary to collect the runtime characteristics of known webshells. The effectiveness of dynamic detection depends on the number of valid features collected. If there are not enough features, it will lead to a lot of false negatives.

Wang et al. [14] proposed a webshell detection method based on Multi-Layer Perceptron (MLP) neural network. It is preferred to compile the webshell sample to obtain the bytecode, then use the TF-IDF tool to perform feature vectorization of the bytecode sequence according to bi-gram, and finally use the multi-layer perceptron to train and predict the model.

For PHP webshell, Cui et al. [15] extracted static features from PHP source files, and used TF-IDF vector and hash vector to extract dynamic features for opcode opcodes, and used static features and dynamic features together as webshell features. The classification is then performed using a combination of the random forest classification algorithm and the Gradient-Boosted Decision Trees (GBDT) classifier algorithm.

Fang et al. [16] first obtained the opcode sequences of PHP files. Then use the FastText algorithm to train the opcode sequence model and predict the corresponding eigenvalues, and combine the predicted eigenvalues and static features into the features of the sample. Finally, the random forest classification algorithm is used for model training and prediction.

Yang et al. [7] proposed an attack detection technique based on the SVM algorithm. This technology realizes the detection of webshell attack traffic in HTTP traffic by supervising the machine learning model. This technique achieves high precision and recall. After detecting abnormal traffic, the system can locate the webshell based on the traffic information. And remove the backdoor in time to ensure the security and stability of the Web server. So it can also help monitor trends in intrusions and network security.

3 Background

In this section, we present background knowledge related to the Java fileless webshell. We first introduce Java web components, which provide the underlying functionality of web service. Most Java web frameworks are built on these components. Attackers construct fileless webshells by maliciously exploiting certain features of these components. We then categorized Java fileless webshells based on the components used by the attackers.

3.1 Java Web Composition

The web server runs a lot of middleware, such as Tomcat [17], Weblogic [18], etc. These middlewares are composed of multiple Java web components, including Servlet, Filter, and Listener. When developers use these components, they need to define subclasses and override some methods. A Servlet is a program that runs on a web server or application server as an intermediary layer between requests from HTTP clients and a database or application on the HTTP server. It is responsible for processing the user's request, and according to the request generates the corresponding return information and provides it to the user. Usually, functions are implemented in the *service( )* method of the Servlet. The Filter is a supplement to Servlet, used to inspect or modify request headers and data. When a request needs to read, write or execute data in memory, it needs to be judged and filtered by a Filter. Filter decides whether these requests have permission to perform these operations. In program development, the filtering logic is implemented in the *doFilter( )* method. Listeners track

specific events in a web application. This feature allows for more efficient resource management and automated processing based on event status. Through the Listener, developers can automatically perform some actions, such as monitoring the number of online users, counting website visits, and monitoring website visits. The most commonly used Listener subclass is ServletRequestListener, which has two methods, *requestInitialized( )* and *requestDestroyed( )*. The *requestInitialized( )* implements the function of monitoring client requests. The *requestDestroyed( )* is used to destroy resources (such as ServletRequest objects) at the end of the request.

Developers generally use the above components together. As shown in Fig. 1, a request goes through Listener, Filter and Servlet in turn. The Servlet finally processes the client's request and generates a response. The response goes through the above three components in reverse order and is returned to the client. We take the user visiting the www.example.com website as an example to introduce the 6 steps in Fig. 1.

1) The user enters www.example.com in the browser. The web server receives the browser's request and tries to forward it to the corresponding Servlet.
2) The request is monitored by the Listener. ServletRequestListener calls the *requestInitialized( )* method, which performs developer-defined functions, such as counting website visits. Then forward the request to Filter.
3) The Filter calls the *doFilter( )* method to execute the filtering logic. A typical *doFilter( )* method body is usually divided into two parts, the first half checks the request and the second half checks the response. At this point, *doFilter( )* executes the first half and then sends the request to the servlet.
4) The servlet calls the *service( )* method, generates the code of the www.example.com web page and encapsulates it in the response. Then forward to Filter.
5) Filter executes the second half of *doFilter( )* to check whether the response conforms to the rules defined by the developer. Then forward to the Listener.
6) ServletRequestListener executes the *requestDestroyed( )* method to destroy resources. The response is then returned to the browser.

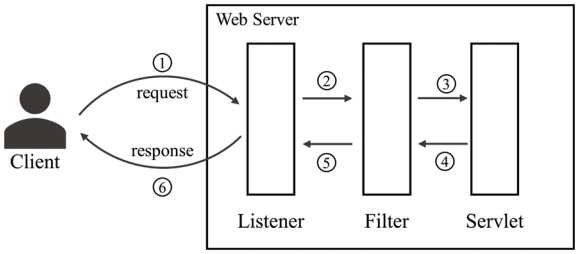

Figure 1: The relationship between Listener, Filter, and Servlet

Fileless webshell attacks take advantage of these components. By dynamically registering a new Filter (or other components) or injecting malicious instructions into an existing Filter (or other components), the attacker makes the Filter allow him to access the data in the memory of the web server. As long as there is a Filter available, an attacker can conduct remote attacks, whether it is command injection or data access. In addition, since the web server is connected to the database and the authentication system in the network, after the intrusion, the attacker can more easily carry out lateral penetration and obtain the permissions of multiple hosts.

3.2 Java Fileless Webshell

Java fileless webshells rely on specific Java web components to perform malicious behavior. According to the different components utilized, Java fileless webshells can be divided into 3 categories, namely Java Web Component-based webshells, Spring framework-based webshells, and Java Instrumentation-based webshells.

3.2.1 Webshells Based on Java Web Component

Java web contains three basic components: Servlet, Filter, and Listener. Java Servlet 3.0 Specification [19] allows these components to be dynamically registered through the ServlFetContext when the web container is initialized. Specifically, an attacker can trace and analyze the method chaining when these components are dynamically registered, and register the components containing malicious code into the web server (e.g., Tomcat) container through the Java reflection mechanism. When the request path contains specific parameters, the malicious component is triggered.

3.2.2 Webshells Based on Spring Framework

The principle of the fileless webshell based on the Spring framework is similar to that of the fileless webshell based on the Java web component. The main difference is that the components that implement the malicious code are replaced by the Spring framework Controller and Interceptor. Taking the Controller-based fileless webshell as an example, the dynamic registration process essentially registers two functions, one is the Controller component itself containing malicious logic, and the other is RequestMapping for address mapping.

3.2.3 Webshells Based on Java Instrumentation

The Java instrument mechanism provides the ability to add bytecode to compiled Java classes. The instrumentation mechanism supports application-independent agents to monitor and assist applications running on the JVM. Such monitoring and assistance include, but are not limited to, obtaining JVM runtime status, replacing and modifying class definitions, etc. So specially crafted agents can intercept requests and responses. Through this mechanism, the attacker injects a specially constructed agent into the Java runtime environment to take over network requests and responses to handle their backdoor requests.

4 Approach

4.1 Overview

In a Java program, a class cannot be used until it has been loaded by the JVM. Therefore, all the features of the Java fileless webshell are in JVM memory, in the form of loaded classes. An intuitive idea is to obtain the bytecode files of all classes loaded in the JVM. When the bytecode file is obtained, the detection of the fileless webshell can be converted into the detection of the traditional file-based webshell.

However, extracting all loaded bytecode files out of the JVM is inefficient and performance-consuming. Therefore, the scope of detection needs to be narrowed, that is, only the bytecode files of suspicious classes are extracted and detected. To achieve this, we implement JShellDetector. As shown in Fig. 2, the process of JShellDetector detecting Java fileless webshell is divided into two steps. The first step is to filter suspicious classes. In a fileless webshell attack, an attacker exploited a vulnerability in the local server to inject a webshell on a remote server into the JVM. We manually selected some

sensitive methods in the JVM and used method probes proposed in this paper to monitor them. When the method probe initializes, the user needs to provide the process id of the protected application. When the webshell tries to call a sensitive method, the method probe activates and marks it as a suspicious class. The bytecode stream of this class will be converted into a bytecode file. The second step is to detect the webshell. A suspicious class is in turn transformed into an intermediate representation of Jimple and a control flow graph. Then taint analysis is used to determine whether the suspicious class is malicious. If there is a path from *sources* to *sinks* on the control flow graph, the suspicious class is considered a webshell. Otherwise, it is considered a benign class.

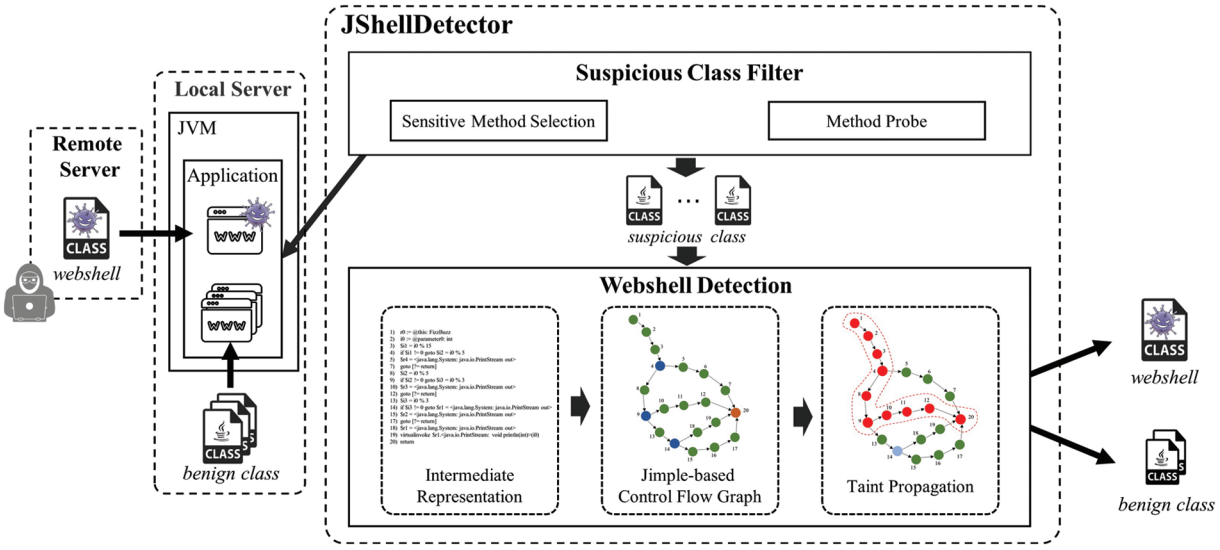

Figure 2: Overview of JShellDetector

4.2 Suspicious Class Filter

The mainstream method for filtering suspicious classes is the blacklist, which is widely used by anti-virus software. Security researchers use strings that contain webshell characteristics as keywords and filter suspicious classes by string matching. However, attackers gradually bypass these keywords by changing the way of constructing webshells, which causes the anti-virus software to fail for the new webshells. In order to maintain the effectiveness of the blacklist, security researchers must constantly update the blacklist. To avoid the drawback of blacklist-based approaches, we propose to filter suspicious classes by monitoring low-level sensitive methods. Because, regardless of how the webshell hides its behavior, some operations must be performed, such as dynamically registering servlets. If we can monitor the status of these operations, we can sense this process when an attacker implants a webshell. To achieve this goal, we propose method probes to monitor these sensitive methods and filter suspicious classes in JVM memory. In this paper, sensitive methods are those that are called when a component is dynamically registered. For example, the fileless webshell based on the Spring Controller component calls *registerMapping( )* to dynamically register, and the Servlet component-based calls *addServletMapping( )* to dynamically register.

4.2.1 Sensitive Method Selection

The selection of sensitive methods needs to meet some conditions. Too many sensitive methods will consume system resources and reduce the access speed of web applications. At the same time, it will lead to more false positives. On the contrary, if the selected sensitive method is insufficient, it will lead to false negatives. So we propose two conditions for selecting sensitive methods: 1) the sensitive method should come from the web framework rather than the method defined by the web application developer; 2) the sensitive method should be as close as possible to the method defined by the developer. For example, if there is a method call sequence (m_1, m_2, m_3), where m_1 is a method defined by the developer, and m_2 and m_3 are methods provided by the framework, m_2 should be selected as a sensitive method. Combining the above selection conditions and manual analysis of Java fileless webshell samples, we selected six sensitive methods for JShellDetector monitoring. The selected sensitive method information is shown in Table 1.

Table 1: Sensitive methods for JShellDetector

No	Class	Method	Description
1	AbstractHandler MethodMapping	*registerMapping*	Register the given mapping.
2	AbstractUrlHandler Mapping	*registerHandler*	Register the specified handler for the given URL paths.
3	Field	*get*	Returns the value of the field represented by this Field, on the specified object.
4	FilterDef	*setFilterName*	Set the name of a filter.
5	StandardContext	*addApplicationEvent Listener*	Add a listener to the end of the list of initialized application event listeners.
6	StandardContext	*addServletMapping Decoded*	Add a new servlet mapping, replacing any existing mapping for the specified pattern.

4.2.2 Method Probe

A method probe is a piece of code that is injected at a specific point in a program. It will automatically execute when a sensitive method is called. We use javassist [20] to implement the method probe and implant it into the target application. Javassist is a class library for dynamically editing Java bytecode. It can define a new class when the java program is running and load it into the JVM, and it can also modify a class file when the JVM is loaded [21]. Javassist enables us to edit class files without understanding bytecode-related specifications. When JShellDetector starts, the user needs to provide the process id of the protected web application. JShellDetector traverses all the classes in the JVM. If the class name is the same as the class name in Table 1, it uses the *insertAfter( )* interface provided by javassist to insert a method probe after the sensitive method. When the method probe is triggered, it will look for the call point of the sensitive method from the stack trace [22], which is a snapshot of a moment in time. Stack trace stores the information of the call stack. Each time a method is called, a method stack will be generated, and it will save the call site information. For example, method A calls

method B, and the context of A is stored on the stack when the call occurs. By traversing the stack trace, the method probe obtains the call site of the sensitive method.

Another function of the method probe is to convert suspicious classes in the JVM into bytecode files. We implement this function through Oracle's *ClassFileTransformer* and *VirtualMachine.attach( )*. The *VirtualMachine.attach( )* method allows us to attach to the JVM running another application by its process id. We use the interface provided by the JVM to process the target value. Then, we create an agent, which inherits from *ClassFileTransformer*. *ClassFileTransformer* provides the ability to convert individual classes to files from the attacher [23]. Method probes convert classes loaded into memory into *<classname> .class* files.

4.3 Webshell Detection

We implement a taint analysis on control flow graphs (CFGs) aimed at determining whether a suspicious class's behavior is malicious [24]. Taint analysis is a technique for tracking and analyzing the flow of taint information through a program. In taint analysis, data of interest (usually from external input to a program) is labeled as tainted data (other data called untainted data). Sources of tainted data are called *sources*. In practice, tainted data usually come from the return values of some methods. By tracking the flow of information related to tainted data, we can know whether they can flow to locations of the program (called *sinks*) and then detect program vulnerabilities [25–27]. *Sinks* are usually some security-sensitive methods.

In this paper, if the suspicious class allows external unauthenticated data to enter some key points of the program, this class will be considered a webshell. That means an attacker can pass instructions to some security-sensitive methods. The *source* we choose is *request.getParameter( )*, which can get the input value of the web page. The *sinks* we choose are *Runtime.exec( )* and *ProcessBuilder( )*. *Runtime.exec( )* executes arguments as commands. *ProcessBuilder( )* creates and modifies operating system processes.

4.3.1 Intermediate Representation

In Section 4.2, we convert the suspicious classes in the JVM into bytecode files. However, Java bytecode contains more than 200 instructions. Direct analysis of bytecode files requires processing all these instructions, making taint analysis difficult. Therefore, we convert the bytecode into an intermediate representation that is easy to analyze. Soot [28] is a state-of-the-art Java bytecode manipulation and optimization framework, providing a set of intermediate representations that can be analyzed and transformed, including Jimple, Grimple, Baf, and Shimple. They represent different levels of abstraction and purpose of use, respectively. We choose Jimple as the intermediate representation for taint analysis. Jimple is a typed three-address intermediate representation [29]. We use Jimple as our intermediate representation because of its three features: 1) Jimple retains some Java data types. 2) Each statement of Jimple includes up to three variable operations, which splits Java's advanced features (e.g., reflection or annotation) into small units. 3) Jimple remains statement characteristics. Compared with bytecode files, each statement of Jimple can encapsulate multiple instructions. These features allow JShellDetector to maintain high accuracy while reducing the complexity of taint analysis.

4.3.2 Jimple-Based Control Flow Graph

CFG is a method of modeling code execution flow, which can display all program execution paths within the method in a graph structure. A CFG is a directed graph, $G = (N, E)$. Each node $n \in N$ corresponds to a basic block. Each edge $e = (n_i, n_j) \in E$ corresponds to a possible transfer of control

from the block n_i to block n_j [30]. JShellDetector uses Jimple as the analysis target. Therefore, it builds a CFG with Jimple units (a soot data structure) as the basic block. Each unit contains statement details.

To illustrate the process of generating CFG through Jimple, we choose a toy example, FizzBuzz. As shown in Fig. 3a, FizzBuzz prints every number from 1 to n. But if the number is divisible by 3, 5, and 15, this program outputs Fuzz, Buzz, and FizzBuzz, respectively. Fig. 3b shows the Jimple code transformed by FizzBuzz, and each line represents a unit. We can get a lot of intuitive information from Jimple. For example, *this* is represented as *r0* which is a *Local* Object. Or the argument of the function is explicitly defined in *i0* and its type is *int*. In Fig. 3c, each circle represents a Jimple unit, and the number is the Jimple code line number. There are 4 possible paths from the beginning to the end of the method, which include 3 branching statements (blue circle). These path representations are divisible by 3, 5, 15, or none of them.

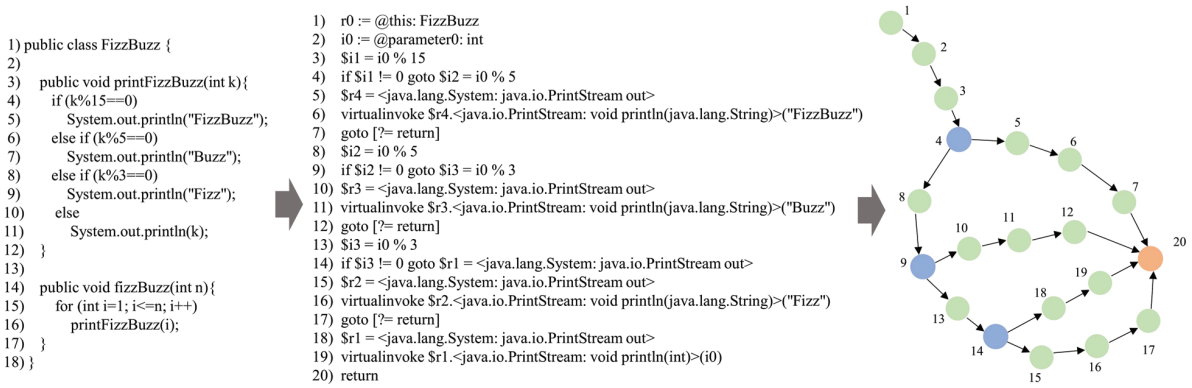

Figure 3: Source code of FizzBuzz conversion to Jimple-based CFG. (a) for FizzBuzz source code, (b) for FizzBuzz's Jimple representation (c) for FizzBuzz's Jimple-based CFG

4.3.3 Taint Propagation

During the taint propagation process, taint data needs to be recorded according to the propagation rules. The taint propagation rule refers to how taints flow between data [24]. Too strict propagation rules lead to under-taint, and too loose propagation rules increase system overhead and lead to over-taint. Table 2 shows the taint propagation rules we use. $\eta\,(val)$ represents the state of the variable *val* in the taint analysis. $\tau \in [t,\ u]$ is the value of $\eta\,(val)$, where t indicates the variable is tainted, and u indicates that the variable is untainted. We use the binary operator $*$ to calculate the taint state, and the calculation rule is as follows:

$$\tau_1 * \tau_2 = \begin{cases} t,\ if\ \tau_1 = t\ or\ \tau_2 = t \\ u,\ if\ \tau_1 = u\ and\ \tau_2 = u \end{cases} \tag{1}$$

The taint propagation rules have a great influence on the accuracy of taint analysis. Therefore, we mainly focus on the rules of variable assignment and function calls. Other operations are generally included in these two operations. For example, for the statement $val = arrray\,[i]$, we need to calculate the taint state of each element of the array, and finally use the calculation result as the taint state of *val*.

Table 2: Taint propagation rules

No	Statement	Condition	Result
1	$val = new\ T()$	/	$\eta\,(val) = u$
2	$val = arrray\,[i]$	$\eta\,(array\,[i]) = \tau_i$	$\eta\,(val) = \tau_1 * \tau_2 \ldots * \tau_n$
3	$val = expr_1 op\ expr_2$	$\eta\,(expr_1) = \tau_1$, $\eta\,(expr_2) = \tau_2$	$\eta\,(val) = \tau_1 * \tau_2$
4	$val = obj\,.x$	$\eta\,(obj) = \tau_1$ $\eta\,(x) = \tau_2$	$\eta\,(val) = \tau_1 * \tau_2$
5	$val = obj\,.f\,()$	$\eta\,(obj) = \tau$	$\eta\,(val) = \tau$
6	$val = obj\,.f\,(a_1,\ \ldots, a_n)$	$\eta\,(a_i) = \tau_i$	$\eta\,(val) = \tau_1 * \tau_2 \ldots * \tau_n$

5 Evaluation

In this section, we evaluate the effectiveness of JShellDetector and compare it with only two tools. We propose the following standard research questions. These questions can help us to comprehensively evaluate the performance of JShellDetector in the detection of Java fileless webshell.

- **RQ1. Effectiveness:** Is JShellDetector able to detect Java fileless webshells? What is the reason for the false negative?
- **RQ2. Comparison with other tools:** Does JShellDetector perform better than other tools? What is the reason?
- **RQ3. Resource Consuming:** Is JShellDetector consuming excessive system resources?

5.1 Experimental Setup

JShellDetector is implemented on JDK 8 (build 1.8_301) of Oracle. To simulate a real Java fileless webshell attack, we need some open-source tools to help us. First, we used an open source network range, SecExample [31], as the attacked web application. A cyber range is a controlled, interactive technological environment where cybersecurity professionals can learn how to detect and mitigate cyberattacks using the same kind of equipment used at work [32]. SecExample provides a simulation environment for CVE-2017-18349 [33]. This is a deserialization vulnerability in the open source software fastjson-1.2.24. The marshalsec [34] is used to provide Java Remote Method Invocation (RMI) services. This service allows the JVM of one server in the same network to access Java classes on another server.

The experiments were performed on two AMD Ryzen 7 4800H CPU@2.9 GHz and 16 GB of RAM on Linux Ubuntu 20.04TSL. One acts as a protected server and the other acts as a malicious server controlled by the attacker. They need to be in the same network. As shown in Fig. 4, SecExample and JShellDetector are deployed on the protected server. Marshalsec is deployed on the malicious server and all test cases are stored. In experiments, we first exploit the CVE-2017-18349 vulnerability to gain permission to execute commands on the protected server (details can be obtained from [33]). We then use the protected server to request the webshell test case from the malicious server's RMI service. Finally, the RMI server returns the webshell test cases on the malicious server to the protected server over the network. At this point, one round of testing is completed. All test cases are tested in the same way.

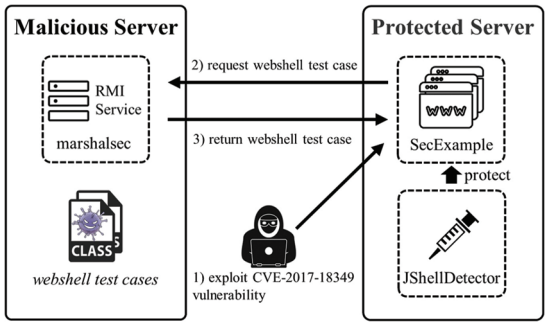

Figure 4: Overview of the experimental environment

5.2 Dataset

Since the research on Java fileless webshell is in its infancy and lacks public datasets, we collect test cases provided by security researchers from the Internet to complete this experiment. We made the dataset public on GitHub.

As shown in Table 3, we collect a total of 35 test cases. According to the web component used by webshell, these test cases are divided into Java web component-based and Spring framework-based. Test cases based on Java web components are further divided into Filter-based, Listener-based and Servlet-based. Test cases based on the Spring framework are further divided into Controller-based and Interceptor-based. For many of these test cases, the attack logic of some of the test cases is the same, only the components used are different. As for Java Instrument-based fileless webshell, we did not collect suitable test cases.

Table 3: Test case dataset

Type	Category	Count
Java web component	Filter	11
	Listener	7
	Servlet	9
Spring framework	Controller	5
	Interceptor	3

5.3 Results and Analysis

5.3.1 Answering RQ1: Effectiveness

To systematically evaluate the effect of JShellDetector, we recorded the results of each step (suspicious class filter and webshell detection). The suspicious class filter step is used to determine whether the method probe accurately captured the call of the webshell to the sensitive method. The webshell detection step means that after the suspicious class is converted into a bytecode file, the taint analysis accurately determines whether it is a webshell.

Table 4 gives the results of each test case. The experimental results show that in the suspicious class filter step, all 35 test cases were successfully captured by JShellDetector, and the detection rate is 100%. As for the webshell detection step, 27 test cases were detected and the detection rate was 77.1%.

We analyze the causes of each failure in detail. There are two reasons for the failure of JShellDetector, namely string encryption and taint analysis failure. Table 5 displays the failed test case corresponding to each reason.

Table 4: Experiment results of each test case

Type	Category	No	Name	JShellDetector		Copagent	Memory Shell
				Suspicious class filter	Webshell detection		
Java web component	Filter	1	addFilter	√	√	√	√
		2	addfilter	√	√	√	√
		3	filterMem	√	×	√	√
		4	memfilter	√	√	√	√
		5	AddFilter	√	√	√	√
		6	FilterShell	√	×	×	×
		7	Frain	√	√	√	√
		8	AddTomcatFilter	√	×	×	×
		9	FilterBasedBasic	√	√	√	√
		10	FilterBasedWithoutRequest	√	√	√	√
		11	FilterBasedWithoutRequestVariant	√	√	×	×
	Listener	12	AddListener	√	√	√	√
		13	addlistener	√	√	√	√
		14	listener	√	√	√	√
		15	memlistener	√	√	√	√
		16	AddListener	√	√	√	√
		17	LRain	√	√	√	√
		18	AddTomcatListener	√	×	×	×
	Servlet	19	AddServlet	√	√	√	√
		20	addservlet	√	√	√	√
		21	memservlet	√	√	√	√
		22	AddServlet	√	√	√	√
		23	Srain	√	√	√	√
		24	AddTomcatServlet	√	×	×	×
		25	ServletBasedBasic	√	√	√	√
		26	ServletBasedWithoutRequest	√	√	√	√
		27	ServletBasedWithoutRequestVariant	√	√	×	×

(Continued)

Table 4: Continued

Type	Category	No	Name	JShellDetector		Copagent	Memory Shell
				Suspicious class filter	Webshell detection		
Spring framework	Controller	28	Evil	√	√	×	×
		29	InjectToController	√	√	×	×
		30	InvisibleShell	√	√	×	×
		31	ReController	√	×	×	×
		32	ControllerBased	√	√	√	√
	Interceptor	33	AddController	√	×	×	×
		34	TestInterceptor	√	√	×	√
		35	AddInterceptor	√	×	×	×

Table 5: Test case for JShellDetector detection failure

Failure reason	Test case number
String encryption	9, 19, 24, 34, 36, 38
Taint analysis failure	3, 7

String encryption is a common approach used by attackers to hide malicious behavior, which is a big challenge for webshell detection. In the dataset we collected, 17.1% of the test cases are encrypted. This technique erases the semantic information of the webshell and makes the code of the attack vector no longer satisfy the Java syntax rules. This means that JShellDetector cannot convert the encrypted webshell to an intermediate representation, which in turn causes taint analysis to fail. As for webshell detection in other languages, existing studies mostly use statistical feature analysis [35] or learning-based methods [36] to solve this problem. However, the Java fileless webshell currently lacks sufficient samples for these methods.

For test cases 3 and 7, the attacker implements malicious logic by manipulating the bytecode. In the real world, ClassWriter object is widely used to manipulate bytecode. Fig. 5 shows an example of using ClassWriter. In Fig. 5a we use the *visit( )* method to define a class named TreeNode, and then add two member variables (i.e., left and right) and a public method (i.e., *print( )*) to it. Fig. 5b is the code generated by the instruction in Fig. 5a. In some test cases, the attacker generates the real attack vector using a manner similar to Fig. 5. However, the target of JShellDetector analysis is that the instruction is based on the ClassWriter rather than the real attack vector generated at runtime, which leads to false negatives. Fortunately, the number of such test cases is not large, accounting for only 5.7%.

In general, JShellDetector can effectively capture the sensitive behavior of attackers injecting fileless webshells, and most malicious behaviors of webshells can be detected through taint analysis.

```
ClassWriter cw = new ClassWriter(0);
cw.visit(V1_8, ACC_PUBLIC,"clazz/TreeNode", null, "java/lang/Object",null);
cw.visitField(0,"left","Lclazz/TreeNode;",null,null);
cw.visitField(0,"right","Lclazz/TreeNode;",null,null);
cw.visitMethod(ACC_PUBLIC,"print","()V",null,null);
cw.visitEnd();
```

(a)

```
package clazz;

public class TreeNode {
    TreeNode left;
    TreeNode right;
    public void print() {}
}
```

(b)

Figure 5: An example of using ClassWriter to generate Java code. (a) is the instruction based on ClassWriter (b) is the generated code

5.3.2 Answering RQ2: Comparison with Other Tools

To our knowledge, only two open source tools, i.e., Copagent [37] and MemoryShell [38], claim to automate the detection of Java fileless webshells. The Copagent extracts all classes in the JVM, and filters suspicious classes by package name, class name, interface name, and annotation. Then Copagent judges whether the behavior of the suspicious class matches the features of the Java fileless webshell. The author of Copagent proposed three features, namely, loading or modifying Java classes, no corresponding bytecode file in the web server, and no registration in the configuration file. The Copagent outputs the risk level of each suspicious class according to the feature matching result, which is processed manually. MemoryShell is optimized in Copagent. On the one hand, his author adds protection against fileless webshell attacks, and on the other hand, optimizes method call analysis.

As shown in Table 6, the performance of MemoryShell and Copagent is similar. MemoryShell captured 23 fileless webshell attacks, which accounted for 65.7% of the total test cases. Copagent detected 22 fileless webshells, accounting for 62.9% of the total test cases. The performance of both tools is lower than the 77.1% detection rate of JShellDetector, which demonstrates again the effectiveness of our proposed tool. Further analysis, we found that the 23 samples detected by MemoryShell included 21 test cases based on the Java web component and only 2 test cases based on the Spring framework, accounting for 77.8% and 25% of the dataset, respectively. This means that MemoryShell performs poorly on Spring framework-based webshells. Similarly, Copagent only captures 1 webshell based on the Spring framework, and the performance is equally bad. For JShellDetector, the detection rates are 81.5% and 62.5% on Java web component-based test cases and Spring framework-based test cases, respectively. Overall, JShellDetector outperforms MemoryShell and Copagent both on each type and overall.

Table 6: Detection rate on different types of test cases

Type	JShellDetector	Copagent	MemoryShell
Java web component	81.5%	77.8%	77.8%
Spring framework	62.5%	12.5%	25%
Overall detection rate	77.1%	62.9%	65.7%

To figure out the reasons for this, we analyzed the differences between the three tools in detail. The main difference between MemoryShell and Copagent lies in two points. The first difference is that MemoryShell uses whether there is a class file in the corresponding ClassLoader directory as the criterion for judging a webshell (e.g., if there is class A in memory, but there is no *A.class* file

on the server, MemoryShell considers A is a webshell). By analyzing the content of the decompiled Java bytecode, Copagent takes whether there are dangerous keywords as the criterion for judging the fileless webshell. The second difference is that MemoryShell specifically adds detection conditions for Interceptor-based webshells. It blacklists *org.springframework.web.servlet.HandlerInterceptor*, which is a class that must be used to construct an Interceptor-based webshell. On the contrary, Copagent lacks the processing logic for Interceptor-based webshells.

In fact, MemoryShell, which uses the existence of class files in the corresponding ClassLoader directory as the only criterion for judging, is very easy to cause false negatives. Although the fileless webshell will not leave traces on the hard disk, if the attacker injects by exploiting the traditional file upload vulnerability in the real attack scenario, as long as the corresponding class file exists, the MemoryShell will be invalid. From the experimental results, the advantage of Copagent compared to our detection tool is that it can detect fileless webshells with encrypted strings. This is because Copagent directly lists *javax.crypto* as a dangerous keyword, and most fileless webshells with encrypted strings call this class for encryption and decryption. However, these two tools are based on the blacklist mechanism for the suspicious class filter. This means that Copagent and MemoryShell cannot detect malicious behavior that does not inherit dangerous parent classes or implement specific interfaces. JShellDetector does not have these problems and has better performance on the fileless webshell based on the Spring framework.

5.3.3 Answering RQ3: Resource Consuming

The additional resource consumption of the system is an important indicator for evaluating the usability of a tool. In order to reduce the pressure on the web server, JShellDetector uses offline analysis to detect webshells. Only method probes consume web server resources. We counted the resource consumption of the test case, and the additional memory resources consumed are all less than 1% of the cyber range SecExample.

6 Conclusion and Future Work

Webshell is a common method for persistent attacks with injection vulnerabilities. As the cybersecurity arms race escalates, traditional file-based webshells have struggled to escape anti-virus software. Attackers are looking for more stealthy solutions. Due to its invisible characteristics, Java fileless webshell makes it challenging for the traditional detection method, and is being used more and more to attack web applications. To address this problem, we propose JShellDetector, a detector based on JVM monitoring and taint analysis. Compared with other blacklist-based or matching-based approaches, our approach has a higher detection rate. In order to systematically evaluate the effectiveness of JShellDetector, we selected other two Java fileless webshell detection tools for comparison. The experimental results show that the detection rate of JShellDetector is about 11% higher than that of these two tools. In addition, the collected test cases are publicly accessed which can help future researchers.

Although JShellDetector performs well on test cases, we must recognize that it still has some limitations. First, the number of test cases is not sufficient, so we cannot determine the performance of JShellDetector on more complex webshells. The lack of test cases also limits the techniques we can use. Approaches based on machine learning or statistical features do not work well on small samples. Therefore, collecting test cases will continue in our future work. Second, JShellDetector cannot handle encrypted strings. Although there are some existing works to detect webshells with encrypted strings based on statistical features, our collection of Java fileless webshells is not sufficient. In future work,

we will try to extract statistical features of traditional file-based webshells and then migrate them to fileless webshell detection.

Funding Statement: This work was supported by the National Natural Science Foundation of China under Grant Number 62001055.

Availability of Data and Materials: Experimental dataset is freely available at https://github.com/java-security/webshelldataset.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] OWASP Top 10 2021 team, *Welcome to the OWASP Top 10-2021*, 2021. [Online]. Available: https://owasp.org/Top10.

[2] D. A. Kindy and A. K. Pathan, "A survey on SQL injection: Vulnerabilities, attacks, and prevention techniques," in *Proc. of 2011 IEEE 15th Int. Symp. on Consumer Electronics (ISCE)*, Singapore, pp. 468–471, 2011.

[3] A. Hannousse and S. Yahiouche, "Handling webshell attacks: A systematic mapping and survey," *Computers & Security*, vol. 108, pp. 1–26, 2021.

[4] WatchGuard Threat Lab, *Internet Security Report-Q4 2020*, 2020. [Online]. Available: https://www.watchguard.com/wgrd-resource-center/security-report-q4-2020.

[5] J. Astle, K. Budnick, S. Clark, A. Dider, B. Donohue *et al., 2022 Threat Detection Report*, 2022. [Online]. Available: https://redcanary.com/resources/guides/threat-detection-report.

[6] W. Kang, S. Zhong, K. Chen, J. Lai and G. Xu, "RF-AdaCost: WebShell detection method that combines statistical features and opcode," in *Proc. of Int. Conf. on Frontiers in Cyber Security*, Tianjin, China, pp. 667–682, 2020.

[7] W. Yang, B. Sun and B. Cui, "A webshell detection technology based on HTTP traffic analysis," in *Proc. of Int. Conf. on Innovative Mobile and Internet Services in Ubiquitous Computing*, Matsue, Japan, pp. 336–342, 2018.

[8] G. You, H. Marco-Gisbert and P. Keir, "Mitigating webshell attacks through machine learning techniques," *Future Internet*, vol. 12, pp. 1–12, 2020.

[9] TIOBE Software BV, *TIOBE Index for July 2022*, 2022. [Online]. Available: https://www.tiobe.com/tiobe-index.

[10] T. Zhu, Z. Weng, L. Fu and L. Ruan, "A web shell detection method based on multiview feature fusion," *Applied Sciences*, vol. 10, no. 18, pp. 6274–6290, 2020.

[11] A. Kurniawan, B. S. Abbas, A. Trisetyarso and S. M. Isa, "Classification of web backdoor malware based on function call execution of static analysis," *ICIC Express Letters*, vol. 13, no. 6, pp. 445–452, 2019.

[12] W. Huang, C. Jia, M. Yu, K. Chow, J. Chen *et al.,* "Enhancing the feature profiles of Web shells by analyzing the performance of multiple detectors," *IFIP International Conference on Digital Forensics*, vol. 589, pp. 57–72, 2020.

[13] Y. Tian, J. Wang, Z. Zhou and S. Zhou, "CNN-webshell: Malicious Web shell detection with convolutional neural network," in *Proc. of 2017 VI Int. Conf. on Network, Communication and Computing (ICNCC 2017)*, New York, NY, USA, pp. 75–79, 2017.

[14] Z. Wang, J. Yang, M. Dai, R. Xu and X. Liang, "A method of detecting webshell based on multi-layer perception," *Academic Journal of Computing & Information Science*, vol. 2, pp. 81–91, 2019.

[15] H. Cui, D. Huang, Y. Fang, L. Liu and C. Huang, "Webshell detection based on random forest–gradient boosting decision tree algorithm," in *Proc. of 2018 IEEE Third Int. Conf. on Data Science in Cyberspace (DSC)*, Guangzhou, China, pp. 153–160, 2018.

[16] Y. Fang, Y. Qiu, L. Liu and C. Huang, "Detecting webshell based on random forest with FastText," in *Proc. of the 2018 Int. Conf. on Computing and Artificial Intelligence (ICCAI 2018)*, New York, NY, USA, pp. 52–56, 2018.
[17] Apache Software Foundation, *Apache Tomcat*, 2022. [Online]. Available: https://tomcat.apache.org.
[18] Oracle Corporation, *Oracle WebLogic Server*, 2022. [Online]. Available: https://www.oracle.com/java/weblogic.
[19] IBM Corporation, *Java Servlets 3.0*, 2022. [Online]. Available: https://www.ibm.com/docs/en/was-liberty/base?topic=features-java-servlets-30.
[20] S. Chiba, "Javassist—a reflection-based programming wizard for java," in *Proc. of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Vancouver, British Columbia, Canada, vol. 174, pp. 21–26, 1998.
[21] S. Chiba, "Load-time structural reflection in java," in *Proc. of European Conf. on Object-Oriented Programming*, Sophia Antipolis and Cannes, France, vol. 1850, pp. 313–336, 2015.
[22] P. Derakhshanfar, X. Devroey, A. Panichella and A. V. Deursen, "Botsing, a search-based crash reproduction framework for java," in *Proc. of the 35th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, Melbourne, Australia, pp. 1278–1280, 2020.
[23] S. Nusbaum, *Dumping embedded Java classes*, 2018. [Online]. Available: https://www.trustedsec.com/blog/dumping-embedded-java-classes.
[24] P. Ferrara, L. Olivieri and F. Spoto, "BackFlow: Backward context-sensitive flow reconstruction of taint analysis results," in *Proc. of Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, New Orleans, LA, USA, pp. 23–43, 2020.
[25] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan and O. Weisman, "TAJ: Effective taint analysis of web applications," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 87–97, 2009.
[26] Y. Ren, J. Zhao and C. Zhang, "The propagation strategy model of taint analysis," in *Proc. of the 4th Int. Seminar on Computer Technology, Mechanical and Electrical Engineering (ISCME)*, Chengdu, China, vol. 1486, no. 4, pp. 1–7, 2020.
[27] X. Zhang, X. Wang, R. Slavin and J. Niu, "Condysta: Context-aware dynamic supplement to static taint analysis," in *Proc. of 2021 IEEE Symp. on Security and Privacy (SP)*, San Francisco, CA, USA, pp. 796–812, 2021.
[28] P. Lam, E. Bodden, O. Lhotak and L. Hendren, "The soot framework for java program analysis: A retrospective," in *Proc. of Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, Galveston Island, TX, USA, vol. 15, no. 35, pp. 1–8, 2011.
[29] A. Bartel, J. Klein, M. Monperrus and Y. L. Traon, "Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot," in *Proc. of ACM SIGPLAN Int. Workshop on State of the Art in Java Program Analysis*, New York, NY, USA, pp. 27–38, 2012.
[30] S. Olin, "Control-flow analysis in scheme," in *Proc. of ACM SIGPLAN'88 Conf. on Programming Language Design and Implementation (PLDI)*, Atlanta, Georgia, USA, vol. 23, no. 7, pp. 164–174, 1988.
[31] tangxiaofeng7, *SecExample*, 2021. [Online]. Available: https://github.com/tangxiaofeng7/SecExample.
[32] M. M. Yamin, B. Katt and V. Gkioulos, "Cyber ranges and security testbeds: Scenarios, functions, tools and architecture," *Computers & Security*, vol. 88, pp. 1–26, 2020.
[33] National Vulnerability Database, *CVE-2017-18349 Detail*, 2017. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-18349.
[34] mbechler, *marshalsec*, 2021. [Online]. Available: https://github.com/mbechler/marshalsec.
[35] Z. Pan, Y. Chen, Y. Chen, Y. Shen and X. Guo, Webshell detection based on executable data characteristics of PHP code, *Wireless Communications and Mobile Computing*, vol. 2021, pp. 1–12, 2021.
[36] Y. Wu, Y., Sun, C. Huang, P. Jia and L. Liu, "Session-based webshell detection using machine learning in web logs," *Security and Communication Networks*, vol. 2019, pp. 1–12, 2019.
[37] LandGrey, *copagent*, 2021. [Online]. Available: https://github.com/LandGrey/copagent.
[38] su18, *MemoryShell*, 2021. [Online]. Available: https://github.com/su18/MemoryShell.