Towards the Static Detection of Compromised Software Components by Topological Anomalies

Oscar Z. de Paiva, Wilson V. Ruggiero, Marcos A. Simplicio Jr.
Computer Engineering Department
Polytechnic School – Universidade de São Paulo
São Paulo, Brazil
e-mail: {ozpaiva | wilson | msimplicio}@larc.usp.br

***Abstract*—Open-source components are critical assets for reducing software development costs and fostering innovation throughout the globe. However, they introduced serious attack venues in modern software supply-chains, as attackers have been able to inject malicious code into these components with relative ease through various means. Such attacks have increased over the past five years and remain frequent despite the various protection mechanisms proposed to prevent or mitigate them. To address this issue, this paper presents a novel protection mechanism aimed at detecting malicious code injected into open-source components and applications via supply-chain attacks. Due to the relevance of technology in corporate environments, we selected Java as the focus of our study. The proposed mechanism is based on a topological static analysis approach: it extracts topological features of uncontaminated programs, and associates these features to normal (i.e., non-malicious) connections of security-critical methods to program parts, and with each other via these parts. These associations are then used to identify the presence of malicious code as topological anomalies in compromised programs. Preliminary results, obtained from a partial implementation of the technique, indicate high detection accuracy (ranging from 86.3% to 99.98% in a publicly-available dataset), and a small number of false-positive classifications – illustrating that the proposed technique is robust and useful in practice. The paper also discusses some potential limitations of our proposal, as well as possible improvements to address them.**

Keywords-Open-Source Security; Supply Chain Attacks; Malware Detection

I. Introduction

Most of the code of modern software applications consists of direct or indirect third-party dependencies, especially Open-Source Components (OSCs) [1]. This demonstrates how essential OSCs have become in reducing software development costs and promoting innovation around the world. However, the ubiquity of these components has unfortunately opened new attack venues for malicious actors to compromise systems via the Software Supply Chains (SSCs) on which these depend. Indeed, in the last decade, attackers have been able to infiltrate communities involved in OSC development, or compromise systems used by them, to inject malicious code and "trojanize" them with relative ease [2]. In 2025, more than one million compromised OSCs were detected by a major vendor [3].

The occurrence of such attacks has not gone unnoticed by cybersecurity specialists, which began to propose various solutions and also mandate/encourage their adoption [4]. Despite those advances, the risk associated with SSC attacks in OSCs remains largely unmitigated.

To address this issue, this article proposes a novel static analysis technique for detecting the types of malicious code most commonly embedded in OSCs through SSC attacks. The forms and behaviors of these types of code were extracted from the "Backstabber's Knife Collection" dataset [5], the most important collection of code disseminated through SSC attacks to date. Essentially, the code corresponds to small excerpts (of no more than a dozen statements). They are also self-contained, i.e., do not exchange data with benign code, and call one or more security-critical methods (mostly related to network access, file I/O and command interpretation) from default APIs of the execution environment. These excerpts implement behaviors typical of *data exfiltrators*, *droppers*, *backdoors*, and *reverse shells*. Hence, they are all associated with the goal of either directly exfiltrating data from a computer system or penetrating it as the first step of an attack.

Our technique is based on a topological approach to static analysis, through which high-level features of a program's control-flow are inspected. These features, while not providing formal guarantees about the program's security properties, enable the identification of topological anomalies that may indicate the presence of malicious code excerpts in it. The analysis begins with a division of the program's call graph into parts, each associated with a cohesive component. We then extract local and global features, i.e., features related to the parts themselves and to how they are connected, respectively. Connections of security-critical methods to the parts, and between the methods through the parts, are then used to select features that may indicate anomalies.

Motivated by the widespread adoption of Java in high-risk enterprise applications, we tested this approach on a synthetic set of Java programs built using XCorpus as basis [6]. Our preliminary results, comprising the evaluation of a detector based solely on local features, indicate that high accuracy is achievable with our technique. Moreover, some of the features were also designed to increase the cost of evasion attacks, which would require the insertion of spurious code that, at least in principle, may be identified by algorithms of dead-code detection. This may also enable the design of threat models based on the percentage of the SSC controlled by the attacker.

The paper is organized as follows. Section II describes our technique, and preliminary results are presented in Section III. Section IV discusses aspects related to the technique's resistance to evasion. Related works are summarized in section

V. Section VI concludes the discussion and presents ideas for future works.

II. Proposed Technique

Our technique relies primarily on a two-step division of the input program's call graph into hierarchically-related parts. Each part produced in the first step corresponds to one of the *packages* to which belong the classes that constitute the program. In the second step, each of these parts is then mapped to an element of the set of *package groups* that constitute the program. For simplicity, those groups are defined in this work by proximity relations between package names, leveraging the naming conventions usually adopted by Java developers.

The partitioning of the call graph into package and package groups provides natural guidance to analyze the topology of the program, following the definitions provided in its code and in the code of its direct and indirect dependencies. Each part contains methods belonging to a single library or to a group of related libraries and, thus, is expected to exhibit characteristics that distinguish it from other parts. Moreover, the relations between parts express how the program relies on its dependencies to operate, and how these dependencies rely on other dependencies, and so on, transitively.

To detect the types of malicious code snippets, we analyze how security-critical methods from standard Java APIs are normally connected to parts of the program and through them. The definition of "normalcy" in this instance relies on features of the program parts and of the relations between them, such that a random insertion of a malicious code snippet into a part that does not exhibit the expected feature values, can be detected as an anomaly. If connections to and between security critical methods are associated with sufficiently rare feature values (and our preliminary results indicate that this is the case), high detection accuracy can be achieved. Moreover, together, these features may contribute to increase the cost of attacks for the difficulty in forging their values – as that would require inserting spurious method calls into the program in a way as to match expected feature values associated with the connection(s) introduced into the parts by the malicious snippet.

To characterize the types of connections to and between security-critical methods associated with the malicious excerpts, it is necessary to establish the distinction between two cases. In the first one (which we denote as Jx), malicious behaviors are implemented with methods from more than one category of functionality from standard APIs. For example, a data exfiltrator can be implemented with methods from the file and network access categories, while a backdoor can be implemented with methods from the network and command interpretation categories. The second case (denoted as NJ) refers to behaviors implemented by a single category of functionality – namely, command interpretation or reflection – that provides access to any other functionality at runtime.

For the Jx type of malicious code snippet, we defined additional subcategories that further characterize the nature of connections between security-critical methods. These definitions are built on the notion of *junction tree*, which is in turn defined as a tree that is contained in the call graph and has, as leafs, the nodes representing security-critical methods from a specific set under consideration. The root of such a tree we call *junction*. Informally, it is the meeting point for the paths starting from each security-critical method, following the edges in reverse order using a breadth-first search algorithm (some examples can be found in the Figure 1). Depending on the height of the tree and the number of distinct package and package groups to which their methods belong (characteristics that denote the topological nature of each junction), four classes of junctions can be defined as follows:

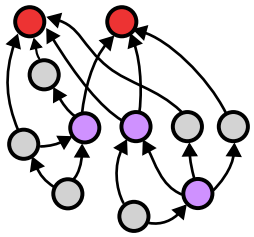

Figure 1. A simple graph illustrating the definitions of junction: relatively to the security-critical methods in red, the vertices in purple represent junctions.

- **J0**: A tree of unitary height, corresponding to malicious functionality implemented inside a single method, from which all calls to security-critical methods are made.
- **J1**: A tree of height greater than one, but whose nodes all belong to a single package containing the calls to the security-critical methods.
- **J2**: A tree of height greater than one whose nodes belong to a single package group but not to a single package. This corresponds to malicious functionality implemented in a single component, from which the calls to security-critical methods are made.
- **J+**: A tree of height greater than one, whose nodes do not belong to a single package group. This corresponds to malicious functionalities implemented by the use of different components, possibly from different libraries or frameworks, from which the calls to security-critical methods are made.

For characterizing program parts and the connections between them, we employ the features defined below. Figure 2 provides examples of all the features. The first four features are, in nature, local (i.e., characterize individual program parts), while the last two are aimed at characterizing more high-level, global aspects of a program.

1) **Incidence**: the number of calls made from a part to methods of a specific Java standard API, expressed both as absolute or, to normalize the feature in relation to the size of the part, relative numbers. Figure 2 (a) shows, relatively to the `net` API (which provides network access functionalities), the incidence of the depicted program part onto the API: 4 in absolute terms, or 0.5 (4/8) in relative terms.
2) **Dispersion**: defined as the number of methods that, within a program part, make calls to a specific standard API. It may also be expressed in absolute or relative terms. Figure 2 (a) shows the dispersion, within a program part, of the

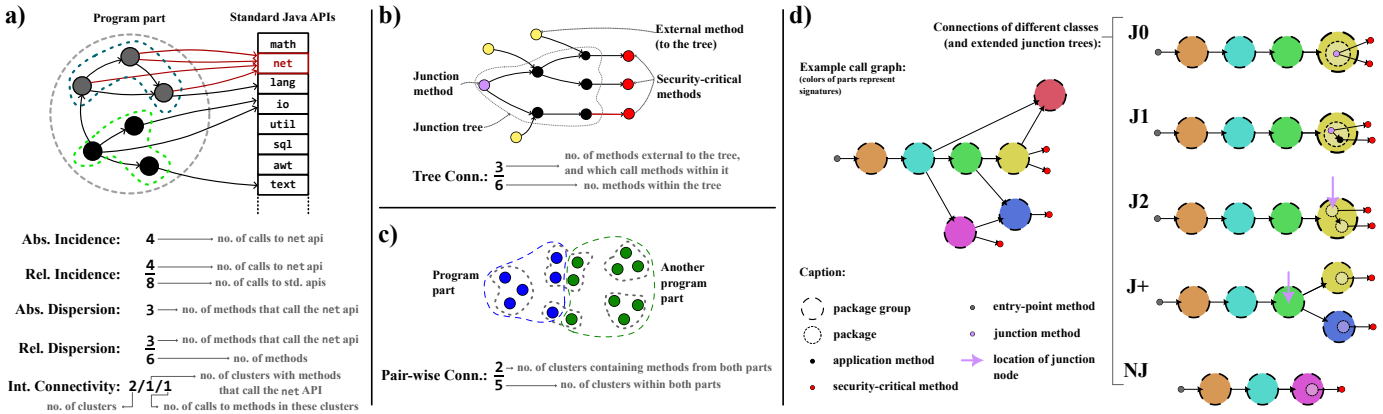

Figure 2. Examples of topological features used to characterize connections of security-critical methods to program parts, and with each other via these parts.

methods that call the `net` API (nodes in gray): in absolute terms, 3, or 0.5 (3/6), in relative terms.

3) **Internal Connectivity**: defined as an indicator of association from a given set of methods in a program part to the rest of the part. It can be obtained by applying a clustering algorithm to the part and then combining: (1) the number of clusters that contain methods from the set; (2) the number of connections between clusters that contain the methods and clusters that do not. Figure 2 (a) shows the internal connectivity of the set of methods that use the `net` API within a program part, expressed as 2/1/1: two clusters in total, one cluster (as defined by the dashed blue line) containing all the methods from the set, and one call to the latter cluster from the other cluster (as defined by the dashed green line).
4) **Tree Connectivity**: expressed as the number of calls made to the methods belonging to a junction tree from methods not in it. Figure 2 (b) shows that, for the depicted junction tree, whose root is the node in purple and which connects the three security-critical methods in red, the tree connectivity is 3, expressing the number of methods that do not belong to the tree (in yellow) but call methods that do. Relatively to the number of elements in the tree, the value of its connectivity is 0.5.
5) **Pair-Wise Connectivity**: expresses how much two program parts are coupled. This feature is defined as a function of both the number of connections between the parts (in absolute or relative numbers), and, once a clustering algorithm is applied to the nodes of both parts, the number of clusters that contain nodes from both parts. Figure 2 (c) shows the pair-wise connectivity between two program parts (depicted in green and blue), which can be expressed as 0.4 (2/5): the number of clusters that contain nodes from both parts (2), among the total number of clusters (5).
6) **Incidence Signatures**: defining the signature of a program part as its set of incidences onto every standard API, a sequence of signatures can be associated with a junction tree as a function of the packages and package groups to which the tree's vertices belong. Likewise, extending the tree backwards into the call-graph, up to the point where the program's expected entry-points are reached, similar signatures can be extracted from the program parts to which the tree is connected (even for the NJ category, if the method that calls the security-critical method is taken as the root of a "virtual" junction tree). In the Figure 2 (d), for an arbitrary graph as depicted, the pictures on the right illustrate, for all possible categories of connections, the path from an entry-point to the security critical methods, and where the junction is located (except for the NJ category). The colors of the program parts represent their incidence signatures, and the typical connections between parts of different signatures represent the most high-level topological property used in our technique.

To build a malicious code snippet detector based on these features, providing both accuracy and robustness against evasion, we rely on the following hypotheses (one for each feature) regarding the normal occurrence of calls to security-critical methods within program parts:

1) The calls are made from parts that have minimal values of incidence onto the APIs containing security-critical methods. In other words, these calls are not exceptions in the parts in which they are made.
2) In these parts, there is a minimum dispersion value associated with calls to the APIs containing the security-critical methods, i.e., there must be at least a few methods that make such calls.
3) There are minimum values of connectivity, within a program part, between the sub-parts in which calls to security-critical methods are made and other sub-parts. In other words, these calls are not made from isolated subparts.
4) For calls associated with junction trees of non-unitary height (i.e., J1, J2 and J+), there are minimum values of connectivity associated with them – i.e., the trees are not isolated in the program parts that contain them.
5) In the paths that begin at the programs' entry-points and lead to a part in which calls to security-critical methods are made, there are minimum values of pair-wise connectivity between each consecutive part involved in the path. This implies that, in each of the parts, a call to the next one in the path is not an exception.

6) From the paths that begin at the programs' entry-points to the parts from which calls to security-critical methods are made, there are expected sequences of signatures among the parts involved in the path. Those sequences can be recognized by Graph Learning Algorithms, such as Graph Neural Networks (GNNs).

A detector combining these features should put an extra burden on SSC attack attempts, which would have to obey high-level topological properties of programs to go undetected. In other words, even though the proposed technique is unlikely to completely prevent the insertion of the malicious excerpts considered in this work (derived from [5]), it is expected to increase the cost of SSC attacks by requiring the adoption of more advanced evasion techniques. In particular, assuming a white-box model (i.e., that attackers know the expected feature values in the program), evasion may be possible by inserting spurious calls into the target parts, having to take extra care to ensure that these insertions are not flagged by dead-code detection techniques. Attackers may also graft, into the call graph of the attacked component, a whole part that naturally meets the expected feature values – for example, by extracting it from some legitimate library and contaminating it with the malicious excerpt. In that case, however, the model of expected signature sequences for analyzing more high-level program features may be helpful in identifying such a malicious implant. Ultimately, threat models that consider the percentage that an attacker controls of the call graph of a program may be defined to analyze security guarantees provided by our method.

III. Preliminary Results

So far, we have implemented a basic version of our detection technique. It was designed to detect only the Jx case of malicious code excerpts, specifically the J0, J1 and J2 cases. The detection of those cases provides validation for the most basic hypotheses underlying our technique: the correlation between incidence and the presence of junctions within the program parts. Moreover, as those cases represent 98% of the instances found in the BKC dataset, they can be considered the most natural choice as the first detection target.

Our implementation relies on the Wala framework [7] to build the call graph of an input program from its bytecode. The framework was configured in a way that reduces computational costs (therefore, possibly not using the most precise static analysis algorithms available in WALA). Two graph processing programs were developed to analyze the call graph: the first extracts from packages and package groups the values of incidences onto standard APIs; and the second searches for natural occurrence of junctions connecting different combinations of security-critical methods, each from three possible package groups from native Java APIs: (1) `net`, which aggregates methods related to network access; (2) `io`, containing methods related to file input and output; and (3) `lang`, which contains methods related to command interpretation and reflection. The combinations were generated in such a way as to represent the largest possible number of implementation possibilities for the malicious behaviors examined in this work, considering Java 1.6 APIs (the latest version compatible with our validation dataset). The data processing step was performed manually, with the help of common spreadsheet software. All the software and data are available in our public repository [8].

The dataset used in our test was XCorpus [6]. Despite being relatively old, this choice was motivated by the fact that it is a curated dataset of compilable and executable programs and program components, which enables the construction of more complete call graphs (as the dependencies of all instances have been resolved). The dataset contains 6694 unique packages, which were manually grouped into 372 package groups by name similarity. The values of minimal incidence were adjusted manually, aiming to maximize detection accuracy considering a scenario of random contamination of packages and package groups. Table I displays the resulting values of minimal incidence associated with junctions connecting security-critical methods within program parts (**J0** or **J0+J1** within packages, or **J0+J1+J2** in package groups), each method belonging to the `io`, `lang` or `net` APIs. The symbols $\theta_I^{(1)}$,$\theta_R^{(1)}$ and $\theta_I^{(2)}$,$\theta_R^{(2)}$ denote the adjusted values of absolute and relative minimal incidences within packages and package groups, respectively. Table I also displays the detector's accuracy and average F_1 score ($\overline{F_1}$) across different synthetic contamination scenarios.

In our tests, we found that the natural occurrence of junctions for classes **J0**, **J1** and **J2** is itself rare. The incidence values displayed in Table I show that, in most cases, junctions of class **J0** are linked to higher relative incidences (and also, to a lesser extent, absolute incidences) than junctions of class **J1** are. These adjusted parameters also show that relative incidences in the packages are generally higher than in package groups, indicating, as expected, that bigger program parts contain code implementing more diverse functionalities. Lastly, these parameters also show that junctions of classes **J2** are associated, with some exceptions, to lower values of relative incidences in comparison to junctions of classes **J1** and **J0**.

The detection accuracies achieved were very high, ranging from 86.3% to 99.98%, and $\overline{F_1}$ ranged between 0.806 and 1. The false positive rate did not exceed 12% in the worst case scenario, indicating the suitability of the method for real-world settings – in which it is very important not to overwhelm human analysts with false alerts. We also performed a simple ablation study, and found that no single parameter contributes to more than 8% to the accuracy in any setting.

IV. Limitations and Resistance to Evasion

Beyond the evasion tactics discussed in Section II, attackers may also employ other tactics, somewhat orthogonal to the former ones, to try to avoid detection. One category of strategies may exploit the limited accuracy of static analysis methods, which causes the construction of incomplete call graphs. An exceptionally thorny case involves Java frameworks, which make heavy use of reflection and annotations to link components at runtime, whereas static analysis tools are not inherently capable of inferring those links. Moreover, as some API implementations are not distributed with applications (for example, because they are part of containers, which serve as

TABLE I. PARAMETERS OF MINIMAL INCIDENCES ASSOCIATED WITH THE PRESENCE OF JUNCTIONS WITHIN PROGRAM PARTS, AND THE DETECTOR'S MINIMAL ACCURACY AND $\overline{F_1}$.

	`io+net` combinations						`lang+net` combinations						`lang+io+net` combinations								
	J0		**J0+J1**		**J0+J1+J2**		**J0**		**J0+J1**		**J0+J1+J2**		**J0**			**J0+J1**			**J0+J1+J2**		
	`io`	`net`	`io`	`net`	`io`	`net`	`lang`	`net`	`lang`	`net`	`lang`	`net`	`lang`	`net`	`io`	`lang`	`net`	`io`	`lang`	`net`	`io`
$\theta_I^{(1)}$	14	7	26	6	–	–	48	13	48	10	–	–	144	24	28	144	24	28	–	–	–
$\theta_R^{(1)}$	4.6%	1.6%	0.6%	1.65%	–	–	8.5%	0.5%	4.2%	0.5%	–	–	33%	2.5%	6.8%	11%	0.3%	1.73%	–	–	–
$\theta_I^{(2)}$	41	17	41	6	62	39	1554	70	583	22	1329	68	11224	725	1618	1329	88	312	1329	40	189
$\theta_R^{(2)}$	2.2%	0.6%	0.1%	1.94%	1.7%	0.35%	9.8%	0.4%	7.8%	0.2%	10%	0.17%	18.6%	1.1%	2.4%	13.6%	0.3%	2.4%	9.1%	0.3%	1%
Acc.	99.2%		97.8%		88.1%		98.3%		95.8%		86.3%		99.98%			99.6%			86.5%		
$\overline{F_1}$	0.993		0.986		0.902		0.985		0.960		0.806		1.00			0.995			0.880		

additional middleware layers), the links established through these APIs are missed by standard static analysis tools. To circumvent these limitations, our technique may be equipped with static representations of frameworks, such as the one found in [9], to allow for more complete call graphs. Another alternative would be to analyze features at runtime, once all connections between program parts are resolved.

Another possible category of evasion tactic involves the obfuscation of package names. In this case, our analysis would require more robust methods to group packages, using not their names but the topological features themselves. Graph similarity techniques may also be used to identify known instances from a dataset of components.

One last evasion technique category of involves aspects of Adversarial Machine Learning, specifically for the possible use of GNNs to identify high-level topological features. In this case, adversarial training and other techniques may be used to make the detection more robust [10].

V. RELATED WORKS

Many methods in the literature to detect malicious code in Java programs and related technologies also guide the analysis by the usage of security-critical methods. The earliest one, to the best of our knowledge, is Jarhead [11], aimed at the detection of malicious applets. For the detection of malicious code in OSCs, [12] investigates simple possible indicators of malicious behavior in JARs. One indicator that the authors conclude to perform is the usage of security-critical methods in a single method. Another related work is Shear [13], which employs program slicing techniques guided by paths between security-critical methods. The slices are then used to train a supervised deep-learning detector. Dapasa [14], in turn, aims to identify piggybacked Android software. The solution relies on noticeable differences between malicious and piggybacked applications in the usage of security-sensitive APIs, guiding their analysis by the concept of sensitive subgraphs. Finally, [15] employs Class-Dependency Graphs to identify disjoint dependency regions in the applications under analysis, noting that isolated regions that make singular use of security-critical methods may be identified as malicious.

Despite sharing similarities with our work, none of these studies analyze high-level topological features of programs, as we do, to guide an unsupervised detection approach.

VI. CONCLUSION AND FUTURE WORK

We present a novel technique for the detection of malicious code embedded in (Java) OSCs through SSC attacks. We follow an anomaly detection approach, using local and high-level topological features to distinguish normal occurrences of connections between security-critical methods and program parts, and between those methods through program parts. The technique was implemented partially, and the tests conducted so far indicate that it can achieve high accuracy considering common code excerpts observed in SSC attacks [5].

As future work, we intend to implement the technique in its entirety, and validate it with a dataset containing more recent versions of Java programs. We also plan to implement and test some of the robustness improvements discussed in Section IV.

REFERENCES

[1] The Linux Foundation, "A summary of census ii: Open source software application libraries the world depends on", 2022, Accessed: 2026-04-16. [Online]. Available: https://www.linuxfoundation.org/blog/blog/a-summary-of-census-ii-open-source-software-application-libraries-the-world-depends-on.

[2] Synopsys Inc., "2025 open source security and risk analysis", 2025, Accessed: 2026-02-19. [Online]. Available: https://www.blackduck.com/resources/analyst-reports/open-source-security-risk-analysis.html.

[3] Sonatype Inc., "Open source malware at the gate: The evolving software supply chain attack surface", 2026, Accessed: 2026-04-16. [Online]. Available: https://www.sonatype.com/state-of-the-software-supply-chain/2026/open-source-malware.

[4] M. Ohm and C. Stuke, "Sok: Practical detection of software supply chain attacks", in *Proceedings of the 18th International Conference on Availability, Reliability and Security*, ser. ARES '23, Benevento, Italy: Association for Computing Machinery, 2023.

[5] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks", in *Detection of Intrusions and Malware, and Vulnerability Assessment*, Cham: Springer International Publishing, 2020, pp. 23–43.

[6] J. Dietrich, H. Schole, L. Sui, and E. Tempero, "Xcorpus - an executable corpus of java programs", *Journal of Object Technology*, vol. 16, no. 4, 1:1–24, Aug. 2017.

[7] J. C. S. Santos and J. Dolby, "Program analysis using wala (tutorial)", in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022, Singapore, Singapore: Association for Computing Machinery, 2022, p. 1819.

[8] O. Paiva, W. Ruggiero, and M. Simplicio, "Towards the static detection of compromised software components by topological anomalies (repository)", 2026, Accessed: 2026-04-21. [Online]. Available: https://github.com/ozpaiva/static-topol-analysis.

[9] M. Chen, T. Tu, H. Zhang, Q. Wen, and W. Wang, "Jasmine: A static analysis framework for spring core technologies", in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22, Rochester, MI, USA: Association for Computing Machinery, 2023.

[10] E. Tabassi, K. J. Burns, M. Hadjimichael, A. D. Molina-Markham, and J. T. Sexton, "A taxonomy and terminology of adversarial machine learning", *NIST IR*, vol. 2019, no. 1-29, p. 1, 2019.

[11] J. Schlumberger, C. Kruegel, and G. Vigna, "Jarhead analysis and detection of malicious java applets", ser. ACSAC '12, Orlando, Florida, USA: Association for Computing Machinery, 2012, pp. 249–257.

[12] P. Ladisa, H. Plate, M. Martinez, O. Barais, and S. E. Ponta, "Towards the detection of malicious java packages", in *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, ser. SCORED'22, Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 63–72.

[13] Y. Zeng et al., "Wolf in sheep's clothing: Shearing the camouflage of malicious java components in maven", *IEEE Transactions on Software Engineering*, vol. 51, no. 10, pp. 2847–2863, 2025.

[14] M. Fan et al., "Dapasa: Detecting android piggybacked apps through sensitive subgraph analysis", *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 8, pp. 1772–1785, 2017.

[15] K. Tian, D. Yao, B. G. Ryder, G. Tan, and G. Peng, "Detection of repackaged android malware with code-heterogeneity features", *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 1, pp. 64–77, 2020.