information

Review

An Analytical Review of the Source Code Models for Exploit Analysis

Elena Fedorchenko *,†, Evgenia Novikova †, Andrey Fedorchenko † and Sergei Verevkin †

St. Petersburg Institute for Informatics and Automation of the Russian Academy of Sciences (SPIIRAS), St. Petersburg Federal Research Center of the Russian Academy of Sciences (SPC RAS), 39, 14th Liniya, 199178 St. Petersburg, Russia; novikova@comsec.spb.ru (E.N.); fedorchenko@comsec.spb.ru (A.F.); vka@mil.ru (S.V.)

* Correspondence: doynikova@comsec.spb.ru

† These authors contributed equally to this work.

Abstract: Currently, enhancing the efficiency of vulnerability detection and assessment remains relevant. We investigate a new approach for the detection of vulnerabilities that can be used in cyber attacks and assess their severity for further effective responses based on an analysis of exploit source codes and real-time detection of features of their implementation. The key element of this approach is an exploit source code model. In this paper, to specify the model, we systematically analyze existing source code models, approaches to source code analysis in general, and exploits in particular in order to examine their advantages, applications, and challenges. Finally, we provide an initial specification of the proposed source code model.

Keywords: software code analysis; exploit; semantic model; feature; graph clustering; cyber threats; software and hardware vulnerabilities; zero-day vulnerabilities; security assessment

check for updates

Citation: Fedorchenko, E.; Novikova, E.; Fedorchenko, A.; Verevkin, S. An Analytical Review of the Source Code Models for Exploit Analysis. *Information* **2023**, *14*, 497. https://doi.org/10.3390/info14090497

Academic Editor: Krzysztof Szczypiorski

Received: 29 June 2023
Revised: 2 August 2023
Accepted: 23 August 2023
Published: 8 September 2023

1. Introduction

Currently, there is a lot of research and many commercial products in the field of cyber attack detection and security monitoring based on signature-based, rule-based, and heuristic methods. However, the number of successful targeted cyber attacks that exploit previously unknown (zero-day) vulnerabilities continues to grow. This indicates the lack of acceptable theoretical and practical solutions in this area. Thus, enhancing the efficiency of vulnerability detection and severity assessment remains relevant. We investigate a new approach to solve the problem of detecting vulnerabilities that can be used in cyber attacks, and for assessing their severity for further efficient responses. This approach includes vulnerability detection and dynamic assessment of their severity based on the analysis of the exploit's code, and the detection of features of their implementation in real-time. To implement this approach, it is required to:

- Develop a reference semantic model of the exploit's source code;
- Define features of the different attack actions considering the related classes of vulnerabilities and weaknesses;
- Develop methods for detection of the attack actions by using vulnerabilities in the real-time, mapping of the features as outlined in the previous step and the characteristics of the analyzed information system;
- Develop methods for assessing the severity of the vulnerabilities and attack actions.

Implementation of these tasks will allow for:

- Detecting the objective features of vulnerabilities and attack actions as determined based on semantic analysis of the source codes of exploits and construction of patterns of their behavior, i.e., of the effects of maliciously intended algorithms on vulnerable software;

- Detecting the cyber attacks at early stages;
- Defining the metrics for the vulnerability assessment and dynamic assessment of a cyber attack's severity based on the selected features.

We propose to use the source code models as the basis for developing a reference semantic model of the exploit's source code. Thus, in this paper, we focused on the existing research in the area of source code analysis as a whole, the source code of the exploits in particular, and their applications for vulnerability assessments.

The main contributions of this paper are as follows:

- A comparative analysis and systematization of the proposed source code models and methods of source code analysis, including the exploit's source code;
- The introduction of an initial semantic model of the exploit's source code that is used for further definition of features of vulnerabilities and cyber attacks.

The paper is organized as follows. Section 2 describes the research methodology and criteria for analysis of the existing source code models and methods of source code analysis, including the exploit's source code. Section 3 compares this comparative research with other reviews. Section 4 gives a detailed description of the selected approaches and their comparative analyses. The paper ends with a summary of the most important results of the study, an introduction of the developed semantic model, and future research prospects.

2. Methodology for the Literature Review and Analysis

The study of source code analysis methods was based on the recommendations for a systematic analysis of the scientific literature [1], which involves defining research questions, strategies for searching and selecting scientific works, and criteria for including and excluding papers in this study. The key objective of this study is to analyze approaches for source code analysis with a possible assessment of the practical applicability to the problem of intrusion detection and assessment. Thus, the following research questions (RQs) were specified:

RQ1: which information security tasks use source code/exploit code analysis?

RQ2: what models and methods of their generation are used for the source code analysis?

RQ3: what programming languages are covered?

RQ4: what metrics are used for the evaluation of the source code analysis methods?

RQ5: what tools are used to generate source code models?

The formulated research questions allowed for specifying the assessment criteria for the selected research papers. The research paper search strategy was specified considering the research questions and recommendations from [1]. The authors analyzed research papers published in scientific journals. We did not consider papers in conference proceedings, non-scientific journals, commercial documents, presentations, or slides. The search was implemented using bibliographic systems. Thus, in order to form a set of research papers for the analysis, the following steps were taken:

Step 1. Form the set of keywords in English.

Step 2. Search the literature using the selected keywords in electronic databases, e.g., ScienceDirect. The result of this step is the initial dataset of the research papers for analysis.

Step 3. Validate the initial dataset using inclusion and exclusion criteria. The criteria are used to decide if the research paper will be included in the final dataset for analysis.

The authors defined the following keywords:

(source code OR exploit source code) AND (graph model OR model) AND (cyber security).

The following inclusion (IC) and exclusion (EC) criteria were defined:

IC1. The research paper clearly describes the approach to source code analysis and proposes an analysis model. The presented approach is used to solve information security

tasks, and the paper describes the experiment scenarios and datasets used, proposes evaluation metrics, and describes any software solutions that are used or developed.

IC2. The paper is well-structured, and the methods are well described.

IC3. The paper is well written, stylistic, and grammar norms are observed.

EC1. The paper is a survey or contains a comparison of the methods.

EC2. The paper is not suitable for the analysis, as it is not relevant to this research.

EC3. The approach presented is poorly described, and the publication lacks a clear structure and/or is presented in unscientific language.

The general scheme of the source collection procedure is shown in Figure 1. First, 393 research articles were selected using the keywords in the *ScienceDirect* database for the given Subject areas—Computer Science, Engineering, and Decision Sciences for 2019–2024 (the last 5 years). Further, taking into account the inclusion and exclusion criteria, 340 articles were excluded; in particular, seven research papers were excluded as review papers, and the rest were excluded as not suitable for the analysis. As a result, only 53 papers were selected for the comparative analysis.

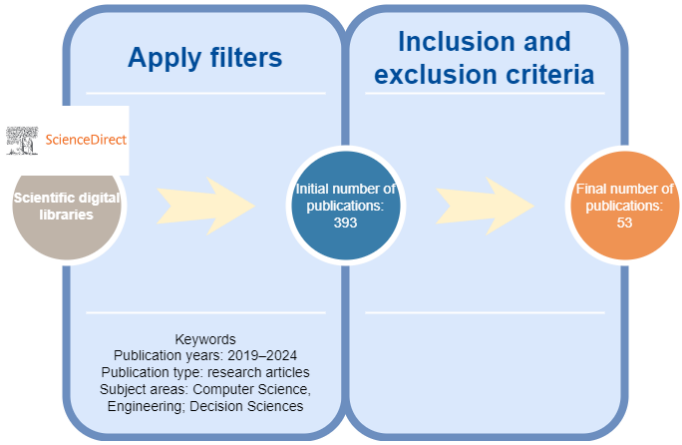

Figure 1. Scheme of the selection process for the research papers, considering the source code analysis for intrusion detection and assessment. All numbers are as of June 2023.

3. Comparison with Other Literature Reviews

There are several reviews in the area of the source code representation for the information security goals that were made within the last 5 years. Some reviews cover various methods for code analysis; thus, paper [2] analyses different vulnerability discovery methods including static code analysis. Paper [3] analyses various approaches to malware analysis, including code analysis, but the review is focused on the applied machine learning and deep learning methods. Paper [4] reviews the detection and analysis of the Linux-based IoT malware, including code analysis methods and models as part of the review.

Some reviews are focused on the analysis of the executable files. In [5], a review of methods for the analysis of the malware in Windows environments (i.e., Portable Executables) is given, while in this review, the methods and models for the source code analysis are considered. In [6], the authors analyze six timestamping approaches for static and dynamic feature sets for Android malware detection.

Some papers review methods of code analysis to detect specific types of attacks. Thus, in [7] the machine learning techniques, and especially deep learning techniques, for the code injection attack detection are reviewed.

Some reviews are devoted to the tools' analysis. Paper [8] compares the static code analysis tools for vulnerability detection from the detected vulnerabilities and weaknesses point of view. The tools for C/C++ and JAVA source code are covered.

Additionally, the adversarial attacks against deep learning models of code should be mentioned. The comparative analysis of the adversarial attacks against deep learning models of code is given in [9].

Thus, from our point of view, there is a gap in reviews of the exploit's source code analysis methods for the cyber security goals, including vulnerability detection and assessment.

4. Models and Methods for the Exploit's Source Code Analysis for the Vulnerability Detection and Assessment

4.1. Source Code Models

There is a set of basic source code models. These models are widely used in such source code analysis and transformation tasks as compilation, code optimization, clone code detection, and code authorship attribution. They are *graph-based models* including Abstract Syntax Tree (AST), Abstract Semantic Graph (ASG), Control Flow Graph (CFG), Program Dependence Graph (PDG), and Code Property Graph (CPG) [10].

Abstract Syntax Tree is a tree which nodes are represented by source code tokens such as operators, variables, and constants. This model reflects the transformation process of source code written in programming language starting from terminal symbols using the grammar of the given language [11]. The definition of the AST is given below [12]:

$$G_A = (V_A, E_A, \lambda_A, \mu_A), \quad (1)$$

where

V_A—the set of nodes;
E_A—the set of directed edges, $E \subseteq (V \times V)$;
λ_A—edge labels of one type: *parent_of*;
μ_A—property assignments of nodes.

An implementation of the AST for the Python language is presented by [13]. We pay special attention to the source code in Python, as the goal of this research is to develop the model for the exploit's static analysis, while Python is one of the most popular languages for the exploit's implementation. Some papers analyze the structure of the .pyc files that can be applied while defining the exploit's source code architecture [14].

An abstract semantic graph is formed from an abstract syntax tree using semantic rules. Unlike an abstract syntax tree, it includes semantic information [11].

A Control Flow Graph (CFG) is an oriented graph where every node is a basic block while every edge connects nodes that can be executed sequentially [15]. The definition of the CFG is given below [12]:

$$G_C = (V_C, E_C, \lambda_C, \varnothing), \quad (2)$$

where

V_C—the set of statements of the programming language, $V_C \subset V_A$, i.e., the CFG nodes are based on the AST nodes;
E_C—the set of edges, where edges define the possible control flow from one statement to another;
λ_C—the edge labels *flows_to*;
$\varnothing$—means that there are no properties for the CFG.

An implementation of the CFG for the Python language is presented by [16].

In [17], the authors generate a set of fingerprints based on the control flow graph of the binary.

A Data Flow graph (https://oneapi-src.github.io/oneTBB/main/tbb_userguide/Data_Flow_Graph.html (accessed on 26 June 2023)) is an orthogonal view of program code. The data flow graph represents data dependencies between a number of operations. It reflects data flows in the program, and is often used in taint analysis to track the information flows.

Program Dependence Graph (PDG) represents data and control dependencies, i.e., it represents the locations in code where variables are used, and resolves function calls. The definition of the PDG is given below [12]:

$$G_P = (V_P, E_P, \lambda_P, \mu_P), \tag{3}$$

where

V_P—the set of nodes;
E_P—the set of edges, where edges define function calls or variable usage, i.e., they link the variable definition and the statements where the variable is used;
λ_P—the edge labels that include the edge label *calls* for the function calls and the edge label *reaches* for the variable usage;
μ_P—the variable name of the variable definition for the variable usage edges.

An implementation of the PDG for the Python language is presented by [18].

In [19], the authors introduce the code property graph (CPG), which integrates AST, CFG, and PDG. It is a directed, edge-labeled, and attributed multigraph [12]. The definition of the CPG is given below:

$$G = (V, E, \lambda, \mu), \tag{4}$$

where

V—the set of nodes;
E—the set of directed edges, $E \subseteq (V \times V)$;
λ—the labels for the edges E: $\lambda \to \sum$, where $\sum$—the alphabet of the edge names;
μ—the properties for edges and nodes: $\mu : (V \bigcup E) \times K \to S$, where K—the set of property keys and S—the set of property values.

In [20], the authors introduce the, static calls graph and the similarity matrix of all possible ways of code execution in the system. The static calls graph represents the relations between the system functions, where each node is a function, and each edge is a function call. The approach proposed by the authors of [20] involves the extraction of the code structure. This structure, in turn, is used to extract the "calling-called" relations, represented as a matrix. A call graph is formed based on this matrix. It is used to generate a similarity matrix. This matrix can be used to discover similar execution paths and to train machine learning models.

In [21], the instruction calls graph is generated based on malicious Android applications. It incorporates all execution paths in terms of the operation codes (opcodes).

Various modifications of these models are currently available. Thus, in [12], the Adversary Controlled Input Dataflow (ACID) tree is constructed based on the code property graph. According to [12], this is "an ordered, rooted, directed, edge-labeled and attributed tree". The definition of the ACID tree is given below:

$$T_{AC} = (V_{AC}, E_{AC}, \lambda_{AC}, \mu_{AC}), \tag{5}$$

where

V_{AC}—the set of nodes, $V_{AC} \subseteq V_A$, i.e., the ACID nodes are based on the AST nodes;
E_{AC}—the set of the directed edges: $E_{AC} \subseteq (V_{AC} \times V_{AC})$;
λ_{AC}—the labels for the edges $E_A C$: $\lambda \to \sum_A C$, where $\sum_A C$—the alphabet of the edge names;
μ_{AC}—the properties for nodes: $\mu_{AC} : V_{AC} \times K_{AC} \to S_{AC}$, where K_{AC}—the set of property keys, S_{AC}—the set of property values.

An example of the ACID sub-tree representing Listing 1 from [12] is given in Figure 2.

Listing 1. Source code.

```
15: \out = $name . intval($page);
    Variable : Assignment : Expression
```

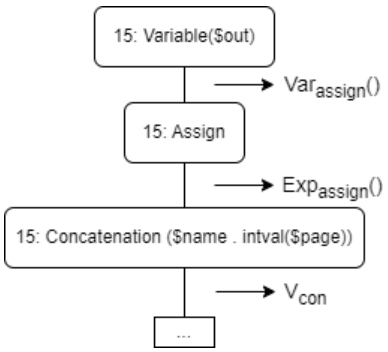

Figure 2. ACID sub-tree for an assignment retrieved from [12].

In [22], the System Call Dependency Graph (SCDG) is introduced for malware analysis and classification. It is constructed based on the binaries. They are used as input to generate symbolic execution traces. SCDG is generated using execution traces. Finally, the SCDGs are used in a supervised machine learning multi-class malware classifier. In [23], the Reflection Guided Static Slicing Diagram (RGSD) is used to detect reflection APIs. It is the state diagram with removed unnecessary paths.

Graph structures also serve as a basis for creating code embeddings. We classify such models as *embedding-based models*, namely, graph-based embeddings. apk2vec [24] is one of the first solutions to construct vector embedding for APK applications. It includes such steps as the generation of CFG with its further decomposition on API dependency graph, Permission Dependency Graph (PDG), and source and sink dependency graph (SGD), extraction of rooted subgraphs from the obtained graphs, and generation of the vector presentation using a skip-gram neural network.

Another widely used representation of the source code is its description using its literal constants. We classify such models as *token-based models*. For example, the source code could be presented as a set of strings [25–27] as a sequence of API calls and its arguments [28]. Depending on the used representation model, different analysis techniques are applied. For example, V. Kalgutkar et al. evaluated the efficiency of the authorship attribution classifiers for strings extracted from various locations of Android APK packages [26], and demonstrated that string-based representation of the code allows outperforming approaches based on analysis of the opcode and bytecode features in authorship attribution task.

In [29], the authors proposed to enrich the string data of the program by including the information about the sequence of the API calls collected using dynamic analysis. Such a solution allowed authors to apply a Bi-LSTM neural network to reveal dependencies between API calls and increase the performance of the malware detection up to 0.9731 in accuracy and 0.9724 in f1 measure.

In [28], the authors add explainability to malware detection by introducing a fusion model of dynamic and static API call sequences based on semantics mapping. The feature vectors are generated considering the contributions of all API calls. They are calculated using the TF–IDF method. The authors use the Mahalanobis distance to measure the similarity of feature vectors of different samples. Furthermore, finally, for malware detection and classification, the authors use the following classifiers: Decision Tree, Random Forest, K-Nearest Neighbor, and Extreme Gradient Boosting. As a result, they reach 97.89% in detection accuracy and 94.39% in classification accuracy.

Binary code analysis is also used in some cases. We classify appropriate models as *byte level models*. It is not aligned with our research goals, but we will consider such models for the completeness of the review. The models of this class that have been gaining popularity

are images [30–34]. Thus, in [34], the space-filling curves (Figure 3a) are used for the construction of the image of the malware behavioral data (Figure 3b).

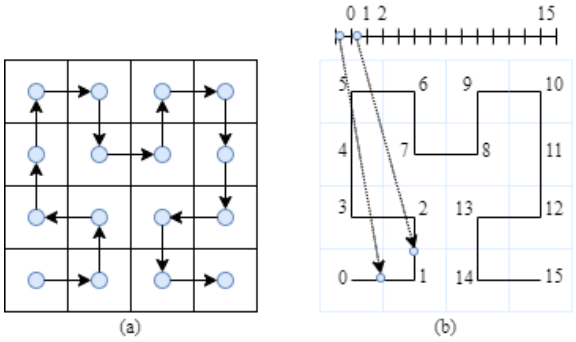

Figure 3. Hilbert space-filling curve traversal (**a**) and mapping data through the space-filling curve (**b**), retrieved from [34].

In [33], the authors use the grayscale images of the malware retrieved from the MalImg dataset [35].

In [33], the traditional learning approach (ResNet50 model) and the transfer learning approach (MCFT-CNN model trained with the knowledge from already trained ImageNet) are used.

Finally, the classification of the analyzed models is provided in Figure 4.

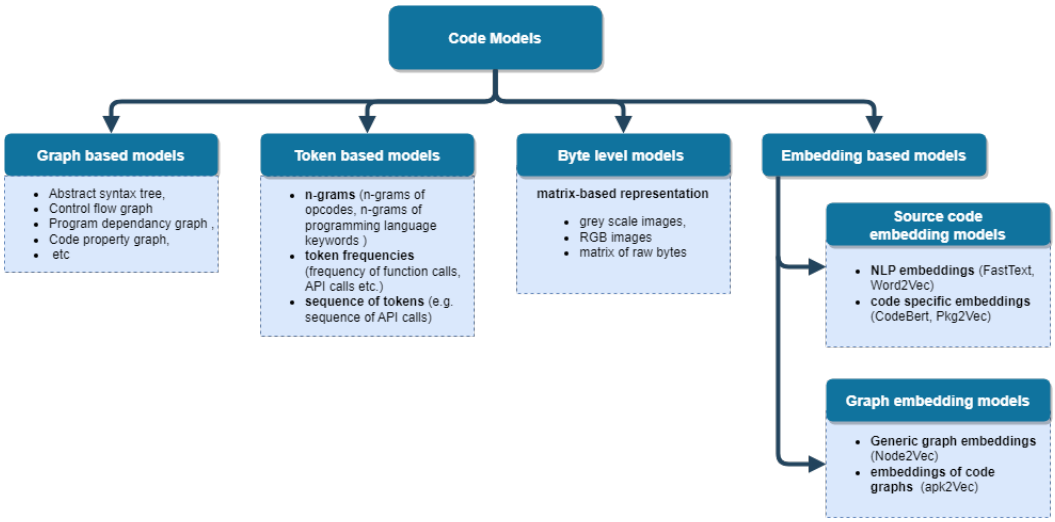

Figure 4. Taxonomy of program code models.

4.2. Source Code Analysis Methods

Various methods based on the proposed models are used for code analysis, including static code analysis, dynamic code analysis, and hybrid analysis. In this study, we mainly consider static analysis methods due to the specifics of the research tasks, but we also consider hybrid (static and dynamic) methods.

Static code analysis based on graph models is quite common. Until recently, inference methods have been rather popular. Static code analysis based on the ACID tree is used in [12]: using PIP definitions and context rules, the authors search possible injection paths in the ACID tree to inject vulnerabilities represented using vulnerability patterns to modify AST and inject data flow patterns to modify AST and generate vulnerable source code.

In [11], the abstract semantic graph is used for static and dynamic analysis. In [17], the authors generate fingerprint sets based on the CFG, and encode them using colors, i.e., arrays of bits where each bit corresponds to the specific class of instructions. Finally,

the obtained fingerprints are compared with the precalculated ones using the similarity measure of the overlap coefficient.

Machine learning methods have become popular in recent years. Several studies apply machine learning and deep learning methods. In some studies, graph-based models are used as the basis for machine learning models. Thus, in [20], the similarity matrix constructed using a static calls graph is used to generate machine learning models. In [22], a supervised machine learning multi-class malware classifier based on the system call dependency graph (SCDG) is used for malware detection and classification.

In [23], the static analysis of the Reflection Guided Static Slicing (RGSS) is used as a preprocessing method to limit the number of paths for the dynamic analysis using machine learning methods.

Other studies use graph-based embeddings to obtain features. In [21], the authors use pseudo-dynamic analysis to generate the instruction calls graph and to extract features for a Deep Neural Network.

Other models are also popular for constructing features for machine learning methods. For example, in [29] the authors use the token-based model to extract features for the Bi-LSTM neural network to reveal dependencies between API calls. In [28], the authors use static and dynamic analysis. They use four different classifiers (Decision Tree, Random Forest, K-Nearest Neighbor, and Extreme Gradient Boosting) trained using features extracted from the token-based models to detect and classify malware.

The methods of the malware detection and classification based on the byte level models, i.e., images, should be outlined [30–34]. Thus, in [34], to extract features from the constructed image, the following algorithms are used: Local Binary Patterns (LBP), Gabor filters, and Histogram of Oriented Gradients (HOG). Finally, the following classification algorithms were used: Random Forest (RF), Support Vector Machine (SVM), and K-nearest Neighbors (KNN).

In [33], the authors avoid the feature extraction process from images of the malware using the traditional learning approach (ResNet50 model) together with the transfer learning approach (a MCFT-CNN model trained with the knowledge from the trained ImageNet).

4.3. Application of the Source Code Analysis Methods in Information Security

Many research works are related to software code analysis using the considered models and methods. Initially, the proposed models and methods were used for the goals of developing and analyzing the functionality of applications, and analyzing programming languages, to ensure correctness and reliability. ASTs are often used for type determination, architecture reconstruction, and call graph extraction [10]. Thus, in [36], the authors use the AST dynamic model to analyze dynamic programming languages. In [37], the authors use AST matching for analysis of the evolution of C source code. The abstract semantic graph is used to develop programs [11], and for the analysis of dynamically typed programming languages in [38]. The Control Flow Graphs are usually used to analyze the flow and control [10,15,39]. The Program Dependency Graphs are used for the program slicing [10,40,41]. The similarity matrix constructed using static calls graph is used to discover similar execution paths [20].

Currently, the main application of source code analysis techniques is to identify software weaknesses and vulnerabilities [19]. In [19], the authors use the code property graph to detect buffer overflows, integer overflows, string format vulnerabilities, etc.

The static and dynamic analysis of executable files is widespread to detect and classify malware. In [17], the authors generate fingerprint sets based on the CFG to detect malicious binaries. In [21], the authors detect malware using pseudo-dynamic analysis. They generate an instruction call graph to construct a feature vector for the deep neural network. In [28], the authors implement static and dynamic analysis of a PE file to detect and classify malware, and to explain why the PE file is recognized as malicious. In [23], the authors apply the static analysis of the Reflection Guided Static Slicing (RGSS) to obtain a sliced app. It allows for limiting the number of paths for the dynamic analysis. The authors use machine

learning methods to detect the reflection API calls in Android apps. The study [22] considers the optimization of malware detection and classification for the malware represented by binary files. Additionally, the image-based malware detection and classification methods should be mentioned [33,34].

Another application of the reviewed methods and models is the generation of vulnerabilities in source code for training purposes. For example, in [12], the authors use the code property graph for this goal. On its basis, the Adversary Controlled Input Dataflow (ACID) tree is built. To inject vulnerabilities, the possible injection paths (PIPs) (i.e., the source code statements that can be refactored to inject vulnerabilities) are searched first. Then, these PIPs are transformed into vulnerabilities using source code patterns specified in the PL/V pattern language.

5. Tools for Code Analysis

In this paper, we analyze the research papers. Most of them name implementation for the experiments but do not develop the code analysis tool [22,28,30,32,34,42–45]. Python is a quite common language for the implementation of the prototypes [21,33]. Thus, in [33], the authors name that the proposed method is implemented in Keras using Tensorflow backend in Python 3.6 environment for the experiments.

In some papers, tools are mentioned but not available in open sources. Thus, in [17], the authors mention that they have developed the tool named Apìcula for malware detection. In [23], the authors mention the tool EspyDroid+ developed for the reflection API calls detection in the Android Apps.

Furthermore, there are just a few papers that provide references to the tool. Thus, in [12] the authors developed the Insecurity Refactoring (https://github.com/fschuckert/insecurity-refactoring (accessed on 26 June 2023)) code refactoring tool, which allows injecting vulnerabilities to the PHP code. To evaluate their tool, the authors used True Positive and True Negative metrics for the found possible injection paths grouped by the corresponding vulnerabilities. In [46], the authors developed the tool (https://github.com/SeUniVr/EtherSolve (accessed on 26 June 2023)) for the construction of the precise CFG graphs based on the analysis of the compiled Ethereum Virtual Machine (EVM) bytecode.

6. Results and Discussion

The summarized results of the comparative analysis are presented in Table 1.

Table 1. Comparative analysis of the research papers.

Ref.	Analysis Model and Method	Programming Languages (Input Data)	Evaluation Metrics	Application
		Graph-based models		
[46]	CFG	Compiled Ethereum Virtual Machine bytecode	Success rate of constructing the CFG	Construction of the precise CFG graphs based on the analysis
[17]	CFG; the set of fingerprints; similarity measure	Binaries (PE)	Jaccard index; Recall; False Discovery Rate (FDR)	Malware detection
[12]	ACID tree. Tree traversal and rules to find place and insert vulnerability patterns.	PHP	TP and FP of the founded places to inject vulnerabilities	Code refactoring
[22]	SCDG; dynamic analysis based on a supervised ML (the methods are not specified)	Binaries	Execution time	Optimizing symbolic execution for malware behavior classification
[23]	Reflection Guided Static Diagram; Reflection Guided Static Analysis (RGSS) to limit number of paths for dynamic analysis based on ML (the methods are not specified)	Android Apps	Number of reflection calls logged and time	Reflection API calls detection

Table 1. *Cont.*

Ref.	Analysis Model and Method	Programming Languages (Input Data)	Evaluation Metrics	Application
		Token-based models		
[43]	Vectorized presentation of the code using Pkg2Vec	Source code decompiled from Dalvik bytecode (Android APK)	Accuracy and loss function	Source code authorship attribution task
[28]	A fusion model of dynamic and static API call sequences based on semantics mapping; TF–IDF method to construct feature vector, Mahalanobis distance to measure the similarity of feature vectors, Decision Tree, Random Forest, K-Nearest Neighbor, and Extreme Gradient Boosting for detection and classification	PE file	TP, TN, FP, FN, precision, recall, f1 score, accuracy	Malware detection and explanation
[47]	Binary header as a sequence of byte code, the CNN neural networks to extract features, GAN to generate adversarial samples of the packed malware. The classification of packed malware is performed using LSTM with attention mechanism	Binaries (PE files and ELF files)	TP, TN, FP, FN, precision, recall, f1 score, accuracy	Packed malware detection
		Byte level models		
[30]	RGB image; convolutional neural network	Source code decompiled form Dalvik bytecode (Android APK)	Precision, recall, f1 measure, accuracy, time for image generation	Malware detection and classification
[31]	RGB image; analysis of the image textures' features using KNN, SVM, NB	Binaries	Precision, recall, f1 measure, accuracy, time for image generation	Identification of the malware variants
[32]	RGB image; analysis of the images using pretrained CNN networks with different architectures: VGG16, VGG19, ResNet50, InceptionV3, MobileNetV2, DenseNet121, DenseNet169	Android bytecode	–	Detection of malicious Android applications
[33]	Grayscale image; ResNet50 model and MCFT-CNN model trained with the knowledge from already trained ImageNet	Grayscale image generated from binaries	Confusion matrix, accuracy, precision, recall, and f measure	Malware classification and uncover
[34]	Space-filling curves to generate an image; LBP, Gabor filters, and HOG to extract features; RF, SVM and KNN for classification	32-bit executable PE samples	Precision, recall, accuracy	Malware classification
		Embedding-based models		
[48]	Code embedding representation based on CodeBERT	–	–	Detecting vulnerabilities in IoT applications
[42]	Code embedding representation based on analysis of word sequence in function; analysis is performed using a Siamese network consisting of BiLSTM and Attention mechanism	C/C++	FNR, FPR, Accuracy, Precision, f measure	Detecting vulnerabilities in IoT applications
[45]	App similarity graph (ASG), which is constructed based on the similarity score calculated for a pair of applications. The similarity score is defined according to the set of functions used by APK. The ASG is used to obtain vector representation of the APK in order to apply classifiers (using node2vec transformation)	APK	Accuracy, f1 measure, AUC	Detection of the malware
[44]	vectorized representation of the Data Flow Graph paths, vectorized representation is constructed using Word2Vec	C/C++ source files	TP, TN, FP, FN, accuracy	Location of vulnerabilities in source code (buffer error (CWE-119) and resource management error (CWE-399))

Table 1. *Cont.*

Ref.	Analysis Model and Method	Programming Languages (Input Data)	Evaluation Metrics	Application
[49]	Combination of the AST and CFG. Graph convolution neural network and the bidirectional recurrent neural network to extract source code features.	JavaScript	TP and FP of the founded places to inject vulnerabilities	Detecting XSS vulnerabilities
[50]	CFG and PDG and program slice, which is a set of Assembler code lines extracted from a binary program, program slice is vectorized and analyzed using LSTM and GRU neural networks	C/C++	FPR, FNR, precision, recall, f1 measure, accuracy	Vulnerability detection
[21]	Instruction call graph based on the opcodes; DNN	Android malware	TP, TN, FP, FN, Precision, Recall, f measure, Accuracy	Malware detection

The review demonstrated that the research direction related to source code analysis is highly relevant to information security. Source code analysis allows for detecting and classifying weaknesses, vulnerabilities, and malware, and for attributing the malware. Despite the number of studies in the area, there are still challenges that should be resolved, including analysis of the obfuscated applications and detection of unknown malware and vulnerabilities. To resolve these challenges, researchers moved their attention to machine learning and deep learning methods. The new challenge of feature construction was revealed. That, in turn, led to multiple researchers in the area of transfer learning based on the images generated from the source code or binaries using different approaches. At the same time, the problem of explainability is almost not covered in the analyzed papers.

The authors noticed just one paper considering the explainability [28], while it could have wide application in information security. Thus, the authors of this paper research the questions of the objective exploits and vulnerabilities measuring. In the analyzed papers, the authors cover only one approach to the automated vulnerability assessment, based on the NLP of the vulnerability descriptions. Such a method is proposed in [51], where the authors categorize vulnerabilities based on their summaries using a taxonomy modeled for industry, and in [52], where the authors use machine learning based on the textual descriptions of the vulnerabilities to predict their CVSS vectors or scores. We assume that the approach based on the source code analysis will allow providing objective and explainable vulnerability and exploit metrics. Thus, in the next section, the authors provide an initial model of the source code for the vulnerability and exploit assessment goals.

7. Conclusions and Future Work

We conducted an analytical review of the research papers related to the models and methods of the source code analysis for the information security goals published in the last five years. Additionally, we compared the conducted review with the existing ones. The analysis was made on the recommendations for the systematic analysis of the scientific literature. As a result, we compared the models and methods proposed in the selected research papers considering the outlined criteria. Finally, we concluded as to why the existing models are not suitable for the goals of the exploit's source code analysis for vulnerability detection. While the graph models could be used to solve the task, already-existing models have disadvantages. Thus, the ASGs do not reflect the code functionality, the abstract semantic graphs in existing implementations lose function call names, and the CFGs may contain vertices that are not functional. Therefore, we propose a new model that is a variation of the semantic graph and combines the control flow graph with the dependency graph of function call names.

The proposed reference semantic model of the exploit's source code is generated based on the CFG, whose nodes are replaced with functions and classes of objects of imported modules (conditionally global names). The nodes of the proposed graph model are specified by the "names" of the source code, i.e., the imported modules and their functions. Such

modules (libraries) make up the standard runtime environment of the code interpreter. The edges of the graph specify the sequence of function calls for such modules. The link from the source node to the destination node represents the use of the corresponding function result as an argument of the destination node function. It can be specified as follows:

$$SG = (V_{in}, E),$$

where

V_{in}—the nodes of the graph that correspond to "names" extracted from the executable code;

E—the edges representing the "names" usage dependencies from the importing modules.

The distinctive features of the model are as follows: a strict adherence to the main code execution route; and a reflection of only functional dependencies between imported names.

An example of the proposed model for the exploit with EDB-ID = 30,688 from the Exploits database is provided in Figure 5: there is a control flow graph on the left and its transformation to the exploit model on the right. In CFG, the nodes correspond to the code blocks. The number in the node represents an offset of the first instruction of each block. The shortest path of the code execution is bold. It represents regular program execution. The nodes beyond the main route represent the blocks that process exceptions and irregular situations.

The generated semantic functional model of the exploit reflects the order and dependencies of calls of imported names. For the provided example, there are names of two functions, namely urlencode() and urlopen(), and one class constructor, namely "Request". The first two calls are made in one block. Therefore, the output of function urlencode() will be unconditionally sent in the constructor of class Request (in case of absence of exceptions). This link is based only on call dependency. Function call urlopen() is made in another block. However, it uses an object of the class Request created earlier. This link is built based on the CFG and functional dependency. Additionally, the code of this exploit contains two more imported modules. However, calls of their modules do not reflect the functional features of code.

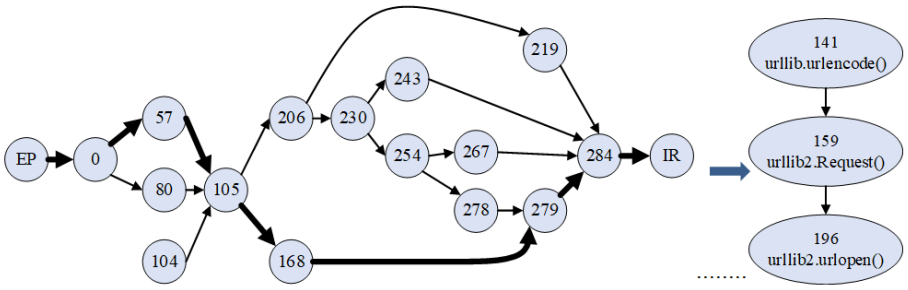

Figure 5. An example of the proposed model for the exploit's source code.

The proposed model of the exploit's source code is required to extract features for vulnerability detection and construction of the objective and explainable vulnerability metrics. It is the core of a new approach under investigation to solve the problem of detecting vulnerabilities that can be used to conduct cyber attacks and assessing their severity for further efficient response.

In the future research, we plan to evolve the proposed model and to develop the method for the vulnerability detection and assessment on its basis.

Author Contributions: Conceptualization, E.F. and E.N.; methodology, E.F. and E.N.; software, A.F. and E.F.; validation, E.F.; formal analysis, E.F. and E.N.; investigation, E.F., E.N., A.F. and S.V.; writing—original draft preparation, E.F., E.N., A.F. and S.V.; writing—review and editing, E.F.; visualization, E.F. and E.N. All authors have read and agreed to the published version of the manuscript.

Funding: This research is being supported by the grant of RSF #23-21-00498 in SPC RAS, https://rscf.ru/en/project/23-21-00498/ (accessed on 26 June 2023).

Data Availability Statement: Data sharing not applicable.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ACID	Adversary-Controlled Input Dataflow
ASG	Abstract Semantic Graph
AST	Abstract Syntax Tree
CFG	Control Flow Graph
CPG	Code Property Graph
EC	Exclusion Criteria
FDR	False Discovery Rate
HOG	Histogram of Oriented Gradients
IC	Inclusion Criteria
KNN	K-Nearest Neighbors
LBP	Local Binary Patterns
PDG	Program Dependence Graph
PE	Portable Executable
PIPs	Possible Injection Paths
RF	Random Forest
RGSD	Reflection-Guided Static Slicing Diagram
RGSS	Reflection-Guided Static Slicing
RQs	Research Questions
SCDG	System Call Dependency Graph
SGD	Sink Dependency Graph
SVM	Support Vector Machine

References

1. Kitchenham, B.A. *Procedures for Performing Systematic Reviews*; Keele University: Keele, UK, 2004.
2. Cui, L.; Cui, J.; Hao, Z.; Li, L.; Ding, Z.; Liu, Y. An empirical study of vulnerability discovery methods over the past ten years. *Comput. Secur.* **2022**, *120*, 102817. [CrossRef]
3. Maniriho, P.; Mahmood, A.N.; Chowdhury, M.J.M. A study on malicious software behaviour analysis and detection techniques: Taxonomy, current trends and challenges. *Future Gener. Comput. Syst.* **2022**, *130*, 1–18. [CrossRef]
4. Madan, S.; Sofat, S.; Bansal, D. Tools and Techniques for Collection and Analysis of Internet-of-Things malware: A systematic state-of-art review. *J. King Saud Univ. Comput. Inf. Sci.* **2022**, *34*, 9867–9888. [CrossRef]
5. Ucci, D.; Aniello, L.; Baldoni, R. Survey of machine learning techniques for malware analysis. *Comput. Secur.* **2019**, *81*, 123–147. [CrossRef]
6. Guerra-Manzanares, A.; Bahsi, H. On the relativity of time: Implications and challenges of data drift on long-term effective android malware detection. *Comput. Secur.* **2022**, *122*, 102835. [CrossRef]
7. Abaimov, S.; Bianchi, G. A survey on the application of deep learning for code injection detection. *Array* **2021**, *11*, 100077. [CrossRef]
8. Kaur, A.; Nayyar, R. A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code. *Procedia Comput. Sci.* **2020**, *171*, 2023–2029. [CrossRef]
9. Li, Z.; Huang, X.; Li, Y.; Chen, G. A comparative study of adversarial training methods for neural models of source code. *Future Gener. Comput. Syst.* **2023**, *142*, 165–181. [CrossRef]
10. Caprile, B.; Potrich, A.; Ricca, F.; Tonella, P. Model centered interoperability for source code analysis. In Proceedings of the STEP 2003, Workshop on Software Analysis and Maintenance: Practices, Tools, Interoperability, Amsterdam, The Netherlands, 22–26 September 2003.
11. Duffy, E. The Design & Implementation of an Abstract Semantic Graph for Statement-Level Dynamic Analysis of C++ Applications. 2011. Available online: https://tigerprints.clemson.edu/cgi/viewcontent.cgi?article=1832&context=all_dissertations (accessed on 26 June 2023).
12. Schuckert, F.; Katt, B.; Langweg, H. Insecurity Refactoring: Automated Injection of Vulnerabilities in Source Code. *Comput. Secur.* **2023**, *128*, 103121. [CrossRef]
13. Astdump 4.3. Available online: https://pypi.org/project/astdump/ (accessed on 26 June 2023).

14. Batchelder, N. The Structure of .pyc Files. Available online: https://nedbatchelder.com/blog/200804/the_structure_of_pyc_files.html (accessed on 26 June 2023).
15. Patterson, E.; Baldini, I.; Mojsilović, A.; Varshney, K.R. Semantic Representation of Data Science Programs. In Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18, International Joint Conferences on Artificial Intelligence Organization, Stockholm, Sweden, 13–18 July 2018; pp. 5847–5849. [CrossRef]
16. Coet, A. StatiCFG. Available online: https://github.com/coetaur0/staticfg (accessed on 26 June 2023).
17. D'Onghia, M.; Salvadore, M.; Nespoli, B.M.; Carminati, M.; Polino, M.; Zanero, S. Apícula: Static detection of API calls in generic streams of bytes. *Comput. Secur.* **2022**, *119*, 102775. [CrossRef]
18. Blais, M. Snakefood: Python Dependency Graphs. Available online: https://github.com/blais/snakefood (accessed on 26 June 2023).
19. Yamaguchi, F.; Golde, N.; Arp, D.; Rieck, K. Modeling and Discovering Vulnerabilities with Code Property Graphs. In Proceedings of the 2014 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 18–21 May 2014; pp. 590–604. [CrossRef]
20. Gharibi, G.; Tripathi, R.; Lee, Y. Code2graph: Automatic Generation of Static Call Graphs for Python Source Code. In Proceedings of the 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), Montpellier, France , 3–7 September 2018; pp. 880–883. [CrossRef]
21. Pektaş, A.; Acarman, T. Learning to detect Android malware via opcode sequences. *Neurocomputing* **2020**, *396*, 599–608. [CrossRef]
22. Sebastio, S.; Baranov, E.; Biondi, F.; Decourbe, O.; Given-Wilson, T.; Legay, A.; Puodzius, C.; Quilbeuf, J. Optimizing symbolic execution for malware behavior classification. *Comput. Secur.* **2020**, *93*, 101775. [CrossRef]
23. Gajrani, J.; Agarwal, U.; Laxmi, V.; Bezawada, B.; Gaur, M.S.; Tripathi, M.; Zemmari, A. EspyDroid+: Precise reflection analysis of android apps. *Comput. Secur.* **2020**, *90*, 101688. [CrossRef]
24. Narayanan, A.; Soh, C.; Chen, L.; Liu, Y.; Wang, L. Apk2vec: Semi-Supervised Multi-view Representation Learning for Profiling Android Applications. In Proceedings of the 2018 IEEE International Conference on Data Mining (ICDM), Singapore, 17–20 November 2018; pp. 357–366. [CrossRef]
25. Zhao, J.; Guo, S.; Mu, D. DouBiGRU-A: Software defect detection algorithm based on attention mechanism and double BiGRU. *Comput. Secur.* **2021**, *111*, 102459. [CrossRef]
26. Kalgutkar, V.; Stakhanova, N.; Cook, P.; Matyukhina, A. Android Authorship Attribution through String Analysis. In Proceedings of the 13th International Conference on Availability, Reliability and Security, Hamburg, Germany, 27–30 August 2018; ARES: Chicago Ridge, IL, USA, 2018. [CrossRef]
27. Alazab, M.; Alazab, M.; Shalaginov, A.; Mesleh, A.; Awajan, A. Intelligent Mobile Malware Detection using Permission Requests and API calls. *Future Gener. Comput. Syst.* **2020**, *107*, 509–521. [CrossRef]
28. Han, W.; Xue, J.; Wang, Y.; Huang, L.; Kong, Z.; Mao, L. MalDAE: Detecting and explaining malware based on correlation and fusion of static and dynamic characteristics. *Comput. Secur.* **2019**, *83*, 208–233. [CrossRef]
29. Li, C.; Lv, Q.; Li, N.; Wang, Y.; Sun, D.; Qiao, Y. A novel deep framework for dynamic malware detection based on API sequence intrinsic features. *Comput. Secur.* **2022**, *116*, 102686. [CrossRef]
30. Naeem, H.; Ullah, F.; Naeem, M.R.; Khalid, S.; Vasan, D.; Jabbar, S.; Saeed, S. Malware detection in industrial internet of things based on hybrid image visualization and deep learning model. *Ad Hoc Netw.* **2020**, *105*, 102154. [CrossRef]
31. Naeem, H.; Guo, B.; Naeem, M.R.; Ullah, F.; Aldabbas, H.; Javed, M.S. Identification of malicious code variants based on image visualization. *Comput. Electr. Eng.* **2019**, *76*, 225–237. . [CrossRef]
32. Yadav, P.; Menon, N.; Ravi, V.; Vishvanathan, S.; Pham, T.D. EfficientNet convolutional neural networks-based Android malware detection. *Comput. Secur.* **2022**, *115*, 102622. [CrossRef]
33. Sudhakar; Kumar, S. MCFT-CNN: Malware classification with fine-tune convolution neural networks using traditional and transfer learning in Internet of Things. *Future Gener. Comput. Syst.* **2021**, *125*, 334–351. [CrossRef]
34. O'Shaughnessy, S.; Sheridan, S. Image-based malware classification hybrid framework based on space-filling curves. *Comput. Secur.* **2022**, *116*, 102660. [CrossRef]
35. Nataraj, L.; Karthikeyan, S.; Jacob, G.; Manjunath, B.S. Malware Images: Visualization and Automatic Classification. In Proceedings of the 8th International Symposium on Visualization for Cyber Security, Pittsburgh, PA, USA, 20 July 2011. [CrossRef]
36. Moses, T.; Barzanti, M. *Static Analysis: A Dynamic Syntax Tree Implementation*; BitBrainery University: London, UK, 2001.
37. Neamtiu, I.; Foster, J.S.; Hicks, M. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. In Proceedings of the 2005 International Workshop on Mining Software Repositories, New York, NY, USA, 15–16 May 2005; pp. 1–5. [CrossRef]
38. Dániel, S.; Gábor, S.; Ádám, L.; Honfi, D. *Graph-Based Source Code Analysis of Dynamically Typed Languages*; Scientific Students' Association Report; Budapest University of Technology and Economics: Budapest, Hungary, 2016.
39. Gold, R. Control flow graphs and code coverage. *Int. J. Appl. Math. Comput. Sci.* **2010**, *20*, 739–749. [CrossRef]
40. Agrawal, H.; Horgan, J.R. Dynamic program slicing. *ACM SIGPLAN Not.* **1990**, *25*, 246–256. [CrossRef]
41. Hsieh, C.; Unger, E.A.; Mata-Toledo, R.A. Using program dependence graphs for information flow control. *J. Syst. Softw.* **1992**, *17*, 227–232. [CrossRef]
42. Sun, H.; Cui, L.; Li, L.; Ding, Z.; Hao, Z.; Cui, J.; Liu, P. VDSimilar: Vulnerability detection based on code similarity of vulnerabilities and patches. *Comput. Secur.* **2021**, *110*, 102417. [CrossRef]
43. Mateless, R.; Tsur, O.; Moskovitch, R. Pkg2Vec: Hierarchical package embedding for code authorship attribution. *Future Gener. Comput. Syst.* **2021**, *116*, 49–60. [CrossRef]

44. Moti, Z.; Hashemi, S.; Karimipour, H.; Dehghantanha, A.; Jahromi, A.N.; Abdi, L.; Alavi, F. Generative adversarial network to detect unseen Internet of Things malware. *Ad Hoc Netw.* **2021**, *122*, 102591. [CrossRef]
45. Frenklach, T.; Cohen, D.; Shabtai, A.; Puzis, R. Android malware detection via an app similarity graph. *Comput. Secur.* **2021**, *109*, 102386. [CrossRef]
46. Pasqua, M.; Benini, A.; Contro, F.; Crosara, M.; Dalla Preda, M.; Ceccato, M. Enhancing Ethereum smart-contracts static analysis by computing a precise Control-Flow Graph of Ethereum bytecode. *J. Syst. Softw.* **2023**, *200*, 111653. [CrossRef]
47. Gao, X.; Hu, C.; Shan, C.; Han, W. MaliCage: A packed malware family classification framework based on DNN and GAN. *J. Inf. Secur. Appl.* **2022**, *68*, 103267. [CrossRef]
48. Mei, H.; Lin, G.; Fang, D.; Zhang, J. Detecting vulnerabilities in IoT software: New hybrid model and comprehensive data analysis. *J. Inf. Secur. Appl.* **2023**, *74*, 103467. [CrossRef]
49. Liu, Z.; Fang, Y.; Huang, C.; Xu, Y. MFXSS: An effective XSS vulnerability detection method in JavaScript based on multi-feature model. *Comput. Secur.* **2023**, *124*, 103015. [CrossRef]
50. Tian, J.; Xing, W.; Li, Z. BVDetector: A program slice-based binary code vulnerability intelligent detection system. *Inf. Softw. Technol.* **2020**, *123*, 106289. [CrossRef]
51. Russo, E.R.; Di Sorbo, A.; Visaggio, C.A.; Canfora, G. Summarizing vulnerabilities' descriptions to support experts during vulnerability assessment activities. *J. Syst. Softw.* **2019**, *156*, 84–99. [CrossRef]
52. Kühn, P.; Relke, D.N.; Reuter, C. Common vulnerability scoring system prediction based on open source intelligence information sources. *Comput. Secur.* **2023**, *131*, 103286. [CrossRef]