A Survey of Source Code Representations for Machine Learning-Based Cybersecurity Tasks

BEATRICE CASEY, JOANNA C. S. SANTOS, and GEORGE PERRY, University of Notre Dame, USA

Machine learning techniques for cybersecurity-related software engineering tasks are becoming increasingly popular. The representation of source code is a key portion of the technique that can impact the way the model is able to learn the features of the source code. With an increasing number of these techniques being developed, it is valuable to see the current state of the field to better understand what exists and what's not there yet. This paper presents a study of these existing ML-based approaches and demonstrates what type of representations were used for different cybersecurity tasks and programming languages. Additionally, we study what types of models are used with different representations. We have found that graph-based representations are the most popular category of representation, and Tokenizers and Abstract Syntax Trees (ASTs) are the two most popular representations overall (*e.g.*, AST and Tokenizers are the **representations** with the highest count of papers, while graph-based representations is the **category** with the highest count of papers). We also found that the most popular cybersecurity task is vulnerability detection, and the language that is covered by the most techniques is C. Finally, we found that sequence-based models are the most popular category of models, and Support Vector Machines (SVMs) are the most popular model overall.

CCS Concepts: • **General and reference** → **Surveys and overviews**.

ACM Reference Format:

Beatrice Casey, Joanna C. S. Santos, and George Perry. 2024. A Survey of Source Code Representations for Machine Learning-Based Cybersecurity Tasks. *ACM Comput. Surv.* 37, 4, Article 111 (March 2024), 39 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Software vulnerabilities are defects that affect a software system's intended security properties [1] which allow attackers to perform malicious actions. As our lives become more attached to technology, software vendors are increasingly pressured into engineering secure software systems, *i.e.*, taking proactive measures of preventing/repairing vulnerabilities prior to deploying the systems into production. There are several practices to address security concerns before software release in each phase of the software development lifecycle. In the requirements phase, *misuse/abuse cases* [2, 3] and *threat modeling* [4] are useful to better understand the security requirements by identifying the potential threats to the system and possible ways to mitigate them. During the design phase, *architectural risk analysis* [5] can be used to assess the likelihood of exploitation of an asset and

Authors' address: Beatrice Casey, bcasey6@nd.edu; Joanna C. S. Santos, joannacss@nd.edu; George Perry, gperry@nd.edu, University of Notre Dame, Notre Dame, IN, USA.

0360-0300/2024/3-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

security tactics and *patterns* [6] can be applied in the design of the software as a proven solution that works under a context. At the implementation phase, *secure code reviews* can be performed to systematically examine the source code with the goal of identifying and fixing vulnerabilities [7]. In the testing phase, *penetration testing* aims to exercise the software in many ways in an attempt to break it and discover vulnerabilities [8] and *static/dynamic analysis* tools can be used to identify potential vulnerabilities in the source code [9, 10].

Although these practices can help improve a software system's security, it can be error-prone and time-consuming for engineers to perform them. For example, finding vulnerabilities in code can be difficult for engineers, especially when they do not know what to look for [7]. With the advances of machine learning (ML), prior works have applied ML techniques to several of these cybersecurity tasks, such as vulnerability detection [11–16], malware detection [17–26] and malicious behavior detection [27]. These techniques are valuable because they can help improve the security of code that is released and speed up otherwise error-prone and time-consuming tasks. For example, a model that detects vulnerabilities prior to deployment would save time, money, and increase the security of the system as a whole, particularly since developers are oftentimes unaware when a vulnerability exists in code until it is exploited or found by security analysts [28].

Machine learning models are unable to understand raw source code, that is, source code in an unprocessed form. As ML models rely on numerical data to adjust the weights necessary for learning, source code must be represented in a form that effectively captures both its *structural* and *semantic* information. This transformation allows the model to understand and learn from the underlying patterns in the code. Thus, **source code representation** is a crucial part during the development of ML-based techniques because different representations will offer different information that the model learns from to perform their task [29].

There are many ways in which source code can be represented, *e.g.*, as an Abstract Syntax Tree (AST) [12, 16, 30–34], Control Flow Graph (CFG) [12, 13, 35–37], tokenized [11, 15, 38–40], *etc.* Although prior works [41–48] have introduced novel source code representations for different cybersecurity tasks, there is no current understanding of what representations exist and are commonly used, as well as the cybersecurity tasks the model is being used for. Furthermore, many of these ML-based techniques are only tested on or made for a particular programming language, and there is no current understanding of the languages that are covered by these techniques. Understanding the available source code representations and what they offer will allow researchers to identify which representation they may want to use based on what task they aim to complete. Additionally, understanding the relationship between cybersecurity tasks and representations will allow researchers to either choose a representation that has previously been used for a particular task, or to test if a different representation would be more suitable.

In this paper, we conduct a ***Systematic Literature Review (SLR)*** by following the guidelines by Kitchenham and Charters [49] to understand the current state of the art of source code representation for ML-based techniques for cybersecurity tasks. We investigate the popularity of certain representations and cybersecurity tasks, the programming languages covered by existing techniques, and the common types of machine learning models used with different representations. We also investigate relationships between representations and cybersecurity tasks. The goal of the study is to allow researchers to understand the gaps in this domain, particularly if there are certain cybersecurity tasks, or languages that are neglected by existing techniques. Additionally, we study and contrast existing representations.

The contributions of this manuscript are: **(1)** an examination of the state-of-the-art of ML-based cybersecurity tasks; **(2)** an investigation of source code representations, and their relationships to cybersecurity tasks and models; **(3)** the identification of the programming languages that are covered/not covered by existing ML-based techniques;

(4) a comparison of the different source code representations. The paper's artifacts, including datasets, code, and additional resources, are available on GitHub: https://github.com/s2e-lab/code-representation-slr.

The rest of this manuscript is organized as follows: Section 2 provides the definition of terms that relevant for understanding our work. Section 3 describes related work. Section 4 explains the methodology of this SLR. Sections 5–9 share the results of this work. Section 10 explains threats to validity, and Section 11 provides a discussion and conclusion of our findings.

2 BACKGROUND

This section discusses core terminology such that the manuscript can be understood by a broader audience.

2.1 Machine Learning for Secure Software Engineering

Along with developing new code of good quality, software engineers are responsible for discovering and resolving bugs, defects, vulnerabilities, and any other issues that could arise in source code. These tasks can be difficult, error-prone, time-consuming, and tedious [7]. Following Kemmerer's [50] definition, a **cybersecurity task** is a task that is aimed at thwarting would-be intruders. Thus, the cybersecurity tasks software engineers work on today involve resolving and identifying code that could allow an attacker to take advantage of a system.

With the recent advances of machine learning, several prior works identified ways in which ML could assist software engineers in these cybersecurity tasks [51]. In particular, given how negatively vulnerabilities and other security-related issues impact companies, researchers started looking into how machine learning can help to mitigate these issues and overall improve the quality of source code that is put out to the public.

2.2 Source Code Representations and Code Embeddings

A **source code representation** captures the source code's *syntax* and *semantics* such that the model is able to learn the key features. There are many ways that a piece of source code can be represented. For example, given the structured nature of source code, prior works captured this structure by representing it as a *graph* [52]. Other works [38, 53, 54] used Natural Language Processing (NLP) techniques on source code in order to leverage technology and knowledge that already exists. These representations offer different information about the source code and thus impact what the model can learn about it. For example, NLP techniques do not offer structural information, but they provide semantic information. Therefore, the model will learn the semantic relationships, but not the structural relationships in the code.

ML models learn from **vector embeddings**, which is a low dimensional way to represent high dimensional data. In the case of learning source code, the embeddings are created from the **source code representation**. The source code representation is the *first* level of abstraction for the original source code. This is what is considered as the *feature extraction* phase. In this phase, the original high-dimensional data are transformed into lower dimensional data, which represents the key features of the data. The feature extraction is aimed at preserving as much of the information about the original data as possible [55]. The vector embeddings are the *second* level of abstraction and are what allows us to perform machine learning techniques on the representations by transforming the representation to the numerical form that machines are able to understand [56]. In this literature review, we focus only on the *first* level of abstraction (*i.e.*, source code representation).

There are a multitude of techniques to create these embeddings. From the realm of NLP work, the embeddings are usually created by using the WORD2VEC model [57], which takes tokens and converts them to numerical vectors. Inspired by this methodology, researchers have found a way to take a graph and create these vector embeddings (GRAPH2VEC [58]). This technique is what is typically used to generate embeddings from the tree or graph structures generated by these source code representations. Graph2vec implements ideas from doc2vec and word2vec, and treats a whole graph as a document and the subgraphs as words [58]. Additionally, other works have also looked at how to create embeddings straight from source code and created techniques such as code2vec [59] and GraphCode2Vec [60].

3 RELATED WORK

To the best of our knowledge, this is the first-of-its-kind SLR that focuses on the representations of source code used in ML-based techniques for cybersecurity tasks. There are a number of papers [61–69] that perform either a systematic mapping study, or a survey of the literature in ML for software engineering, with some papers focusing on vulnerability detection, analysis, or assessment. All of these papers primarily focus on the *ML models* used, but few mention or give detailed descriptions of the representations used in these efforts. Additionally, some of the papers focus only on deep learning techniques [64–66]. Unlike these prior works, our survey paper has its primary focus on the topic of representations used for ML for security-related tasks.

Ghaffarian and Shahriari [70] surveyed techniques used for vulnerability analysis and outlined four main categories that the approaches of these techniques fall under: *software metrics*, *anomaly detection*, *vulnerable code pattern recognition* and *miscellaneous*. Similarly, Nazim *et al.* [63] analyzed deep learning models for vulnerable code detection. Their study examined the different dataset types used to train deep learning models (*e.g.*, synthetic, semi-synthetic, real data, *etc*), the evaluation metrics used to assess performance as well as the different source code representations used. Unlike these prior works, we look beyond vulnerability analysis and detection, and instead look at *all* security-related software engineering tasks. We also have a broader scope because we look at all types of ML models, not just deep learning models.

Wu [71] performed a literature review on NLP techniques for vulnerability detection. While this paper does give a brief overview of different types of representations, its focus is on NLP techniques, particularly NLP models that are focused for code intelligence, such as CodeBERT and CodeXGlue, in the instance of vulnerability detection. Chen and Monperrus [72] perform a similar literature review, investigating word embedding techniques on programs. In this survey, the authors explore different granularities of embeddings from different papers and show visualizations of the different embeddings. We focus on a wide array of security-related tasks, as well as many representations and how they can impact the model's ability to learn vulnerabilities.

Kotti *et al.* [73] performed a tertiary study on ML for software engineering. This paper evaluated 83 reviews, or surveys, on the field of ML for software engineering. While our paper focuses on *security related* software engineering tasks, Kotti *et al.* investigated all ML based software engineering tasks. Additionally, while this paper is a broader analysis of the software engineering tasks that ML covers, our paper focuses primarily on the representations of source code used to perform a security task.

Two papers create a taxonomy for software engineering tasks and ML, with one paper focusing broadly on software engineering challenges for ML systems [74], and the other focusing on software vulnerability detection and ML approaches [75]. Neither of these papers delve into source code representation efforts and what information they offer. However, Hanif *et al.* [75] do mention the importance of representation in the ML pipeline. Unlike these studies, our paper focuses on the representations used, rather than just the models, and focuses on a wide array of security tasks.

Usman *et al.* [76] performed a survey on representation learning efforts in cybersecurity. However, this paper does not focus on the representation of source code, but rather different ML algorithms used for cybersecurity issues, as well as datasets and how industry utilizes these different efforts to improve their cybersecurity. Similarly, Macas *et al.* [77] created a survey on deep learning techniques for cybersecurity. This paper focuses on deep learning techniques to analyze internet traffic. It provides insights and future directions in this area of deep learning to analyze internet traffic for cybersecurity.

4 METHODOLOGY

We followed the guidelines outlined by Kitchenham and Charters [49] to conduct our SLR, which involves three major activities: *planning*, *conducting*, and *reporting* the review. During the *planning* phase, we defined this study's research questions and the search query used to find papers. During the *conducting* phase, we searched three library sources and downloaded all the papers we found into CSV files. We then applied our *inclusion* and *exclusion* criteria in three phases to eliminate papers until we got to the final group of papers that are included in this study. Two reviewers independently read each paper and performed an analysis, extracting the information that is relevant to the research questions we developed in the planning phase. We reviewed and resolved discrepancies to get the final analyses. We calculated the Cohen's Kappa to evaluate the reliability of our evaluation. Our score of 0.97 indicates that we had a near-perfect agreement in our analysis. Finally, during the *reporting* phase, we analyzed our data and organized it so that we could answer the research questions we posed.

4.1 Research Questions

Through this SLR, we aim to answer five research questions.

RQ1: *What are the most commonly used source code representations?*

In this first question, we investigate the source code representations that were used for solving security problems. We aim to understand what source code representations are more popular and compare their trade-offs.

RQ2: *Do certain cybersecurity tasks only or mostly use one type of source code representation?*

In this question, we investigate what source code representations are being used for each cybersecurity task and whether certain representations are favoured for specific tasks. Along with this, we want to investigate why it would be the case that a particular representation is preferred for a task.

RQ3: *What cybersecurity tasks are covered by the techniques that have been created?*

We investigate how these tasks fit into the software development life cycle to identify how these techniques would be used during software development. Furthermore, we describe and find every cybersecurity-related task so that we provide a clearer picture of the task and its importance in the realm of software security.

RQ4: *What programming languages are predominantly targeted by the ML-based techniques for cybersecurity tasks?*

Given the vast number of programming languages used in practice, we investigate what languages the source code analyzed by these techniques is written. This question helps reveal any gaps in the coverage of programming languages by these techniques.

RQ5: *What models are commonly used with different representations?*

In this last question, we study what source code representations are used for different model types. By examining the frequency co-occurrence between models and representations, we aim to understand common trends and preferences in the design of ML-based techniques. This analysis helps identify potential gaps and opportunities for improvement in how models are paired with specific representations.

4.2 Search Method

We used the following search string to find all *primary* studies related to the representation(s) of source code for ML-based cybersecurity tasks: `(“machine learning” OR “deep learning” OR “artificial intelligence”) AND (“security” OR “vulnerability”) AND (“code”)`. While this is a very general search string, which resulted in a total of 67,512 papers, we decided that rather than having a specific string that may miss a category of software security tasks or representations, we would make a general string and manually eliminate any papers that do not meet our inclusion criteria, or fit our exclusion criteria.

We searched three databases to find relevant papers: the ACM Digital Library[1], IEEE Xplore[2], and Springer Link[3]. We also searched nine ML and natural language processing conferences which are considered to be A* conferences by the CORE conference ranker. These nine conferences are: National Conference of the American Association for Artificial Intelligence (AAAI), Conference on Learning Theory (COLT), International Conference on Learning Representations (ICLR), International Conference on Machine Learning (ICML), International Joint Conference on Artificial Intelligence (IJCAI), Advances in Neural Information Processing Systems (NeurIPS, formerly known as NIPS), Association for Computational Linguistics (ACL)[4], Empirical Methods in Natural Language Processing (EMNLP), and International Conference on the Principles of Knowledge Representation and Reasoning (KR). We did not search papers published at the International Joint Conference on Autonomous Agents and Multiagent Systems (previously the International Conference on Multiagent Systems, ICMAS, changed in 2000), the ACM International Conference on Knowledge Discovery and Data Mining, or the IEEE International Conference on Data Mining because these conferences' proceedings were included in the ACM and IEEE libraries.

Table 1. Inclusion and Exclusion criteria

Inclusion Criteria	Exclusion Criteria
I1 Written between 2012- May 2023 **I2** A full paper **I3** Focused on ML for cybersecurity tasks **I4** Contains information regarding the source code's representation	**E1** Duplicated studies **E2** Books, reference work entries, reference works **E3** Position papers, short papers, tool demo papers, keynotes, reviews, tutorials, and panel discussions. **E4** Studies not in English **E5**: Survey/comparative studies.

4.3 Inclusion and Exclusion Criteria

Table 1 lists the inclusion/exclusion criteria applied to the papers in multiple stages in order to eliminate papers irrelevant for this study. We limited our search to papers published between **January 2012** to **May 2023**. Our inclusion criteria focused on ML-based techniques for cybersecurity that involved representing source code. We eliminated duplicate studies, works not in English, and any papers that were not full papers, *e.g.*, books, short papers (*i.e.*, papers with less than five full pages of text, not including references), tutorials, *etc*). We also disregarded any papers that did not represent the source code itself (*e.g.*, papers that dealt with binary files,

[1]https://dl.acm.org
[2]https://ieeexplore.ieee.org
[3]https://link.springer.com
[4]When we got the data for ACL, that includes the Transactions of the Association for Computational Linguistics and the ACL Anthology.

extracting from the Android Manifest, *etc*), as we are interested in only understanding the representation of raw source code.

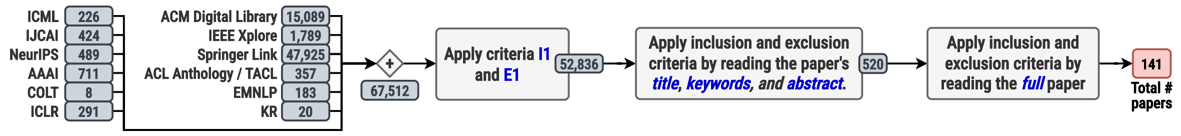

ICML 226
IJCAI 424
NeurIPS 489
AAAI 711
COLT 8
ICLR 291
ACM Digital Library 15,089
IEEE Xplore 1,789
Springer Link 47,925
ACL Anthology / TACL 357
EMNLP 183
KR 20
+
67,512
Apply criteria I1 and E1
52,836
Apply inclusion and exclusion criteria by reading the paper's title, keywords, and abstract.
520
Apply inclusion and exclusion criteria by reading the full paper
141
Total # papers

Fig. 1. Overview of the Three Stages of our Search Process

4.4 Paper Selection

Figure 1 shows the number of papers that made it through each stage of the selection process. We started out with **67,512** total primary studies. We first excluded duplicate studies and studies that were outside the year range of 2012-May 2023 (criterion **I1**), as well as non-full papers (criterion **E1**). This resulted in **52,836** papers. Subsequently, we inspected each paper's title, keywords, and abstract to include/exclude papers based on whether they fit our criteria. After this search, we were left with **520** papers. We then applied the same criteria on these 520 papers, this time by reading the full paper. This left us with the **141** papers that are included in this survey.

4.5 Data Extraction

As we went through the papers, we extracted the key information we were looking for to answer our research questions: the *representation* used , the *cybersecurity task* it was completing, the *programming languages* the technique was designed for or tested on, and the *model type* used in the work. When determining the representation used, we used a *one-paper-to-many-representations* approach based on what the paper claims to be the representation(s) used. That is, we consider every representation that a paper explicitly uses to train / fine-tune their model(s) as a separate, valid representation. However, in the case a paper employs one or more representations solely as internal steps to derive a final representation, then we treat only the final representation as the representation used. For example, if a paper refines an initial control flow graph (CFG) through multiple transformations to produce a regular expression to represent the code, we capture only *regular expression* as the representation being used by the paper.

Besides capturing the metadata above to directly answer our five research questions, we also conducted a comprehensive analysis of the performance of the source code representation(s) . Specifically, we went through the reported results in each paper, extracting the metrics provided. This allowed us to aggregate the data and compare the performance of different source code representations across various cybersecurity tasks.

5 RQ1 RESULTS: WHAT ARE THE COMMONLY USED SOURCE CODE REPRESENTATIONS?

Table 2 summarizes the source code representations used/described by the surveyed papers. Similar to a prior work [64], we organized these representations into four categories: *graph-based*, *tree-based*, *lexical*, and *miscellaneous*. We found that the three most commonly used source code representations are a **tokenizer**, an **abstract syntax tree (AST)** and **code metrics**. We also found two papers [51, 178] that used a slightly modified AST version to represent source code (we denoted them as **AST+** in Table 2). Although more rare, some papers [21, 87, 92] represented code as an **image** analysis, and performed an image analysis of source code to identify patterns that are associated with vulnerabilities or malware. The following subsections give a detailed explanation of what each representation is and the information it carries out to the embeddings.

Table 2. Source code representations, their categories, and frequency of use in the surveyed papers.

	Representation	Papers	#		Representation	Papers	#
Graph-Based	Control Flow Graph (CFG)	[12, 13, 35–37, 78–86]	14	Graph-Based	Component Behavior Graph (CBG)	[27]	1
	Program Dependence Graph (PDG)	[12, 87–93]	8		Component Dependency Graph (CDG)	[27]	1
	Data Flow Graph (DFG)	[13, 37, 78, 80, 84, 94, 95]	7		Contextual ICFG (CICFG)	[26]	1
	Call graph	[20, 22, 25, 35, 79, 96]	6		Contextual Permission Dependency Graph (CPDG)	[26]	1
	Code Property Graph (CPG)	[12, 97–100]	5		Contextual Source and Sink Dependency Graph (CSSDG)	[26]	1
	Interprocedural Control Flow Graph (ICFG)	[18, 19, 26]	3		Crucial Data Flow Graph (CDFG)	[101]	1
	Contextual API Dependency Graph (CADG)	[19, 26]	2		Program Graph	[102]	1
	Contract Graph	[46, 103]	2		Propagation Chain	[104]	1
	Program slices	[14, 105]	2		Property Graph	[45]	1
	Simplified CPG (SCPG)	[106, 107]	2		Semantic Graph	[108]	1
	System Dependence Graph (SDG)	[109, 110]	2		Slice Property Graph (SPG)	[111]	1
	Token Graph	[112, 113]	2		Value Flow Graph (VFG)	[44]	1
	Code Aggregate Graph (CAG)	[114]	1				
Lexical	Tokenizer	[11, 15, 17, 38–40, 43, 53, 54, 82, 86–90, 104, 115–145]	47	Miscellaneous	Code metrics	[96, 116, 132, 133, 146–157]	16
	intermediate code and Semantics-based Vulnerability Candidate (iSeVC)	[48, 158, 159]	3		Code gadgets	[14, 41, 110, 160–163]	7
	source code and Syntax based Vulnerability Candidate (sSyVC)	[48, 158]	2		Image	[21, 87, 92]	3
	Contract Snippet	[47]	1		Opcode Sequences	[164]	1
Tree-based	Abstract Syntax Tree (AST)	[12, 16, 30–34, 37, 78, 80, 85, 88–90, 106, 142, 144, 157, 163, 165–177]	32		Application Information	[23]	1
	AST+	[51, 178]	2		API Calls	[24]	1
	Parse Tree	[179]	1		Regular Expression	[180]	1

5.1 Tree-Based Source Code Representations

Tree-based representations are those that demonstrate the *hierarchical* nature of source code [64].

5.1.1 Abstract Syntax Tree (AST). An **AST** is a tree representation of source code that provides information about code elements (*e.g.*, variables) and their structural relationship [63, 181]. It was the most popular representation (used by **32** papers). Although one paper [168] used CODE2VEC [182] as a way to generate source code embeddings, the basis of their model is an AST; the source code is represented as an AST before the vectors are generated.

Using the information from an AST, models can capture general structural code patterns, since ASTs abstract away the low-level syntax details of the underlying programming language of the code [12]. This reduces learning effort and allows for ASTs to be used for multiple tasks [59]. Embeddings for ASTs can be generated in different ways. Typically, the node and path are what form the embedding, so that the relationship between two nodes can be effectively captured by the embedding [59].

5.1.2 Parse Tree. A **parse tree** represents the hierarchy of *tokens*, *i.e.*, the program's terminal and non-terminal symbols. This data structure is generated by the language's parser [183]. Thus, the nodes represent the derivation of the grammar that yields the input strings. This representation has been used by Ceccato *et al.* [179] to represent SQL queries in order to train a model that detects SQL injection vulnerabilities. Although **parse trees** and **ASTs** both represent source code in a tree structure, their key difference is that ASTs are much *simpler* than parse trees, as they abstract away grammar-related nodes while parse trees retain these tokens and their meanings with respect to their grammar. In the appendix (A.1), we provide an example of a parse tree and AST for the same source code, to demonstrate these differences.

5.1.3 AST+. One paper [178] used a representation that is an enhanced version of an AST (which we denote in our manuscript as **AST+**). That work uses a convention [184] that describes AST nodes in three types: *placeholder*, *API*, and *syntax nodes*. The ASTs are serialized and traversed using depth first traversal, and each node and element is mapped to a vector. Xia *et al.* [51] do not specify the modifications made to the AST, however the paper states that additional edges are added to the AST to capture more semantic and stream information. This representation was used for vulnerability detection [51, 178].

5.2 Graph-Based Representations

Graph-based representations are those that transform source code into a *graph* form, with nodes and edges representing certain characteristics and relationships, respectively, between each code element. Graph-based representations can be embedded using Graph2vec [58], as it is an optimized method to transform the graphs into the low dimensional numerical vectors that the models will learn from. As shown in Table 2, we found **24** different graph-based representations, whose descriptions are provided in the next sections.

5.2.1 Control Flow Graph (CFG). A **CFG** [185] was the most popular graph representation used in **14** papers [12, 13, 35–37, 78–86]. A CFG is a directed graph $g = (V, E)$ with nodes $v \in V$ and edges $e \in E$, where $E \subseteq V \times V$. The set of nodes V represents the *basic blocks* of a program's procedure (*i.e.*, a function/method), while the edge set E represents the control flow between the basic blocks. A basic block is a group of instructions that are executed in order, one after the other. A CFG's edge $e = v_{src} \longrightarrow v_{dst}$ denotes that the program's execution can flow from v_{src} to v_{dst}.

5.2.2 Interprocedural Control Flow Graph (ICFG). An **ICFG** is a variation of the CFG that describes not only *intra-procedural flows* among the basic blocks, but also ***inter-procedural*** ones [19]. An ICFG connects individual CFGs at the call sites in order to represent control flows across procedures. Thus, the ICFG allows the model to understand the control flow of the *whole program*, whereas the CFG allows the model to understand the control flow of a specific *procedure* [186].

5.2.3 Data Flow Graph (DFG). Seven papers used a **Data Flow Graph** (**DFG**) which is a graph $G = (V, E)$, where the nodes $v \in V$ are statements in the source code, and the edge set $e \in E$ are the data dependencies between the nodes. That is, an edge $e = v_{src} \longrightarrow v_{dst}$ indicates that the node v_{dst} uses data that has been defined by v_{src}.

5.2.4 Program Dependence Graph (PDG). PDGs were the second most popular graph-based representation, being used by **8** papers. A **Program Dependence Graph (PDG)** [187] is a directed graph $g = (V, E)$ that shows the *data* and *control* dependencies for each statement in a program's procedure. The set of nodes V in a PDG is partitioned into two types: *statement* nodes V_{stmt} and *predicate expression* nodes V_{pred}. A *statement* node $v_{stmt} \in V_{stmt}$ represents simple statements in a program that are actions to be carried out by a program (*e.g.*, `x = 2;`). A *predicate* node $v_{pred} \in V_{pred}$ denotes statements that evaluate to *true* or *false* (*e.g.*, `x != 2`). The edge set E in a PDG has two partitions: *control dependency* edges E_c and *data dependency* edges E_d. A control dependency edge $e_c = v_{src} \longrightarrow v_{dst}$ indicates that v_{dst} only executes if the predicate expression in v_{src} evaluates to true. A data dependency edge $e_d = v_{src} \longrightarrow v_{dst}$ denotes that v_{dst} uses data that has been defined by v_{src}.

5.2.5 Call Graph. A **call graph** is a directed graph $g = (V, E)$ in which the nodes are the functions/methods in a program, whereas the edges represents caller-callee relationships among program's procedures [188]. An edge $e = v_{src} \longrightarrow v_{dst}$ denotes that v_{src} invokes v_{dst}. These graphs can be of two types: *static* and *dynamic* call graphs. **Dynamic call graphs** give information regarding the procedure calls of a program while it is being executed. It shows the sequence of function/method calls, and the parameters that are passed to each procedure in the sequence. **Static call graphs** only give information about the *potential* execution paths a program can have based on information available at compile time. Thus, a static call graph is not as accurate in reflecting the actual calls in a program, particularly if the program is complex [188]. **6** papers included in this survey used static call graphs [20, 22, 25, 35, 79, 96].

In the appendix (A.2), we provide an example of a CFG, ICFG, DFG, PDG, and call graph to show the different type of information each representation provides for the same code snippet.

5.2.6 System Dependency Graph (SDG). Two papers represented source code using SDGs [109, 110]. An **SDG** is a graph with multiple PDGs connected via the caller-callee relation given by a call graph. SDGs extend PDGs by describing the *inter-procedural* relationships between the program's *entrypoints*[5] and the procedures they call [109]. To connect the PDGs, there are additional nodes and edges which dictate the *actual input parameters* and *actual output values* of a procedure. Every passed argument has an *actual in* node a_i, and a *formal-in* node f_i which are connected by the parameter-in edge $a_i \longrightarrow f_i$. Every modified parameter and returned value has an actual-out node a_o and a formal-out node f_o, which are connected by the parameter-out edge $f_o \longrightarrow a_o$. The formal-in and -out nodes are control dependent on the entry node e and actual-in and -out nodes are control dependent on the call node c. This parameter passing model ensures that interprocedural events of a procedure are propagated by the call sites.

5.2.7 Program Slices. A **program slice** [189] is a subgraph of a PDG or SDG that includes only the nodes that are relevant to a computation at a specific point in the program. This subgraph is computed using a *slicing criterion* $\langle v, p \rangle$ which denotes a variable of interest v at a program point p. These slices can be computed in a *backward* or *forward* fashion. A **backward slice** includes all the nodes that may affect the value of v at the program point p. A **forward slice** includes all nodes that are affected by the variable v at the program point p. Program slices were used by **2** papers [14, 105] to detect vulnerabilities. The slicing criterion is determined by statements in the code that are considered as vulnerable. The statements could also be points where values are changed, which could then lead to an API call being vulnerable. Cheng *et al.* [105] use a PDG and perform forward and backward slicing from the node of interest. It is not specified in [14] whether a PDG or SDG is used, but the same criterion for backwards and forwards slicing are used (*i.e.*, forward slices include statements that are affected by the node of interest, and backward slices include statements that affect the node of interest).

5.2.8 Crucial Data Flow Graph (CDFG). A **CDFG**, introduced by Wu *et al.* [101] is a subgraph of a DFG graph that contains only the *crucial* information from the DFG that could trigger a reentrancy vulnerability in Smart contracts. The *crucial* nodes are variables containing sensitive or critical information, and that have a direct data flow to another crucial node. A CDFG is defined as $CDFG = (V, E)$, where $v \in V$ are the crucial nodes and the edges $e \in E$ represent the data flow relationship. For example, $e = v_{src} \longrightarrow v_{dst}$ indicates that v_{src} and v_{dst} are both crucial nodes, and that there is a data flow between the two variables.

[5]Entrypoints are the functions/methods that starts the program execution, *e.g.*, the `main()` function in C.

In the appendix (A.2), we provide an example of an SDG, a Program Slice, and a CDFG as a demonstration for what information can be expected from each representation.

5.2.9 Program Graph. Wang *et al.* [102] introduced the concept of **program graphs**, which is a directed graph $g = (V, E)$ in which the nodes $v \in V$ can be *statements*, *identifiers* (*e.g.*, function declarations or variables), or *values*. This graph has eight types of edges: *control-flow* edges, *data flow* edges, *guarded by* edges, *computed from* edges, *next token* edges, *last use* edges, and *last lexical use* edges. A *control flow edge* $e_{ctr} = v_{src} \longrightarrow v_{dst}$ indicates that v_{dst} can execute after v_{src}. A *data flow edge* $e_{data} = v_{src} \longrightarrow v_{dst}$ indicates that v_{dst} uses a variable that has been defined by v_{src}. A *guarded by* edge $e_g = v_{src} \longrightarrow v_{dst}$ indicates that v_{dst} only executes if the expression in v_{src} evaluates to true (which is useful to identify operations that may be in the wrong order). A *jump* edge $e_j = v_{src} \longrightarrow v_{dst}$ indicates that v_{dst} has a control dependency from v_{src}. A *ComputedFrom* edge $e_{compFrom} = v_{src} \longrightarrow v_{dst}$ indicates that v_{src} is or contains a variable used in an expression in v_{dst}. A *NextToken* edge $e_{next} = v_{src} \longrightarrow v_{dst}$ indicates that v_{dst} is a successor of (*i.e.*, follows) v_{src}, where v_{dst} and v_{src} are terminal nodes or tokens from the AST. A *LastUse* edge $e_{last} = v_{src} \longrightarrow v_{dst}$ indicates that v_{dst} uses the same variable that is used in v_{src}. A *LastLexicalUse* edge $e_{lastLex} = v_{src} \longrightarrow v_{dst}$ indicates that v_{dst} uses the same variable that is used in v_{src} if v_{src} is an if statement.

5.2.10 Code Property Graph (CPG). Used by **5** papers [12, 97–100], a **Code Property Graph** (**CPG**) is a combination of ASTs, CFGs and PDGs [52]. It was first introduced by Yamaguchi *et al.* [52] specifically as a way to detect vulnerabilities in C/C++ programs using static analysis. The way a CPG is generated is by taking the AST, CFG and PDG of a program, modeling them as *property graphs*, and then these models are jointly combined by connecting statement and predicate nodes. A CPG is formally defined as $g = (V, E, \lambda, \mu)$, which is a directed, labelled, attributed multigraph, with nodes $v \in V$, edges $e \in E$, edge labeling function λ and a property mapping function μ. The set of nodes V in a CPG are the nodes from an AST. The edge set E in a CPG has three partitions: *control flow edges* $E_{cf} \subseteq V \times V$, *program dependency* edges $E_{pd} \subseteq V \times V$, and *abstract syntax tree* edges $E_{ast} \subseteq V \times V$. A *control flow dependency* $e_{cf} = v_{src} \longrightarrow v_{dst}$ indicates that v_{src} can flow to v_{dst} in the next step of the program. A *program dependency* edge $e_{pd} = v_{src} \longrightarrow v_{dst}$ indicates that v_{dst} has a program dependence edge from v_{src}. An *abstract syntax tree* edge $e_{ast} = v_{src} \longrightarrow v_{dst}$ indicates that v_{dst} is syntactically related to v_{src}. The *edge labeling function* $\lambda : E \longrightarrow \Sigma$ assigns a label from the alphabet Σ to each edge in E. The function $\mu : (V \cup E) \times K \longrightarrow S$ applies properties to nodes and edges, where K is the set of property keys and S is the set of property values. Since a CPG is a combination of so many representations, it provides a very robust understanding of code. Other implementations of a CPG enhance it by adding information from a Data Flow Graph (DFG) [63].

5.2.11 Simplified Code Property Graph (SCPG). While code property graphs are able to capture rich semantic and syntactic information, it is also very complex to create. Generating a PDG alone has a complexity of $O(n^2)$. The size of the graphs are also rather large, one example having 52 million nodes and 87 million edges [52]. To solve this issue, two papers implemented a **Simplified CPG** (**SCPG**) [106, 107]. A simplified CPG only uses edges from an AST and a CFG, as data dependence can be approximated from these two graphs. The nodes in the SCPG have two values: the *code tokens*, and the *node type*. Removing the need to generate a PDG greatly reduces the cost of generating this representation, as one would only need to generate the AST and CFG.

5.2.12 Property Graph. A **property graph** [45] is variant of a CPG, defined as $g = (V, E, \Lambda, \Sigma, \mu, \lambda, \sigma)$. Here, the edges and nodes are the same as the CPG. The adjacency function $\mu : E \longrightarrow V \times V$ maps any edge to an ordered pair of its source and destination vertices. The function $\lambda : V \longrightarrow \Lambda$ maps any given vertex to its respective attributes Λ, and $\sigma : E \longrightarrow \Sigma$ is the attribute function for edges, which, just as λ, maps any given edges to its respective attributes Σ.

5.2.13 Code Aggregate Graph (CAG). A **Code Aggregate Graph** (**CAG**) is built from a combination of an AST, CFG, PDG, dominator tree (DT) and post-dominator tree (PDT). A CAG is formally defined as $g = (V, E)$, which is a directed labelled, attributed multigraph, with nodes $v \in V$ and edges $e \in E$ where $E \subseteq V \times V$. The set of nodes V in a Code Aggregate Graph are the nodes from an AST. The edge set E in a Code Aggregate Graph has five partitions: *control flow* edges $E_{cf} \subseteq V \times V$, *program dependency* edges $E_{pd} \subseteq V \times V$, *abstract syntax tree* edges $E_{ast} \subseteq V \times V$, *dominator tree* edges $E_{dt} \subseteq V \times V$, *post-dominator tree* edges $E_{pdt} \subseteq V \times V$. A *control flow dependency* $e_{cf} = v_{src} \longrightarrow v_{dst}$ indicates that v_{src} can flow to v_{dst} in the next step of the program. A *program dependency edge* $e_{pd} = v_{src} \longrightarrow v_{dst}$ indicates that v_{dst} has a program dependence edge from v_{src}. An *abstract syntax tree* edge $e_{ast} = v_{src} \longrightarrow v_{dst}$ indicates that v_{dst} is syntactically related to v_{src}. A *dominator tree* edge $e_{dt} = v_{src} \longrightarrow v_{dst}$ indicates that the operation v_{dst} is dominated by v_{src} and all of v_{src} dominators (*i.e.*, all paths from the entry node to v_{dst} first pass through v_{src}). A *post-dominator tree* edge $e_{pdt} = v_{src} \longrightarrow v_{dst}$ indicates that the operation v_{dst} is post-dominated by v_{src}, meaning that all paths from v_{dst} to the end node must pass through v_{src}. Using a dominator tree and post dominator tree allows this representation to better capture semantic information in source code. This, in turn, allows models to perform better in the task of vulnerability detection. Nguyen *et al.* [114] points out certain information that a CFG and an AST in particular fail to capture, and how a DT and a PDT can better describe these attributes.

5.2.14 Value Flow Graph (VFG). A **Value Flow Graph (VFG)** is similar to a program dependence graph in that is shows the interprocedural program dependence. The edges, just like in a PDG, describe the control flow and data dependency of the program [190]. A VFG $g = (V, E)$ is a directed labelled graph, with nodes $v \in V$ and edges $e \in E$. The set of nodes V are a pair (γ_1, γ_2) in which γ_1 is a node from the pre-directed acyclic graph (DAG) and γ_2 is a node from the post-DAG. Both γ_1 and γ_2 represent the same value. The edge set E in a VFG are a pair (v, v') such that $N(v)$, the node flow graph of v, is the predecessor of $N(v')$, the node flow graph of v', and values are maintained along the connecting edge [191].

The paper [44] that used a VFG uses a special process that selects and preserves feasible value-flow paths to reduce the amount of data needed for training models for path-based vulnerability detection . This makes their method more lightweight than a typical Value Flow Graph would be.

In the appendix (A.2), we provide an example of a CPG, CAG, and a VFG for the same source code to highlight the differences in the type of information provided by each representation.

5.2.15 Component Dependency Graph (CDG). A **Component Dependency Graph** (**CDG**) [27] represents the relationships between the different components in a graph and was created to capture Android app program logic. The component dependency graph is formally defined as $g = (V, E)$, which is a directed labelled graph with nodes $v \in V$ and edges $e \in E$. The set of nodes V in a component dependency graph represent the components of the Android app (*i.e.*, *Activity*, *Service*, or *Broadcast Receiver*). The edge set E in a component dependency graph represent the activation relationship between the components. An edge $e = v_{src} \longrightarrow v_{dst}$ indicates that the component v_{src} could activate the start of lifecycle of the component v_{dst}.

5.2.16 Component Behavior Graph (CBG). A **Component Behavior Graph** (CBG) [27] represents the lifetime or control-flow logic of the permission-related API functions in a Java or Android program, as well as the functions performed on a particular resource for each component. This is the second half of the the Component Dependency Graph, where both of these representations come together to fully describe the Android app. There are four types of nodes, each indicating the type of component at that portion of the graph. The edges connecting the CBG demonstrate the control flow logic between the API functions and sensitive resources.

The component behavior graph $g = (V, E)$ is a directed labelled graph with nodes $v \in V$ and edges $e \in E$. The set of nodes V in a component behavior graph is partitioned into four types: root node V_{root}, lifecycle function nodes V_{life}, permission-related API function nodes V_{prapi}, and sensitive resource nodes V_f. A *start node* $v_{root} \in V_{root}$ represents the component itself. A *lifecycle function node* $v_{life} \in V_{life}$ represent the runtime programming logic.

Each *permission-related API functions node* $v_{prapi} \in V_{prapi}$ denotes a permission-related API function, for example Android's API `sendTextMessage()`. A *sensitive resource node* $v_f \in V_f$ indicates sensitive data that is accessed by a component. The edge set E in a component behavior graph represent the control flow logic of the framework API functions and sensitive resources. A component dependency graph edge $e_{cbg} = v_{src} \longrightarrow v_{dst}$ indicates either that, if v_{src} and v_{dst} are in the same control-flow block, then v_{dst} is executed right after v_{src} with no executions in between, or if v_{src} and v_{dst} are in two continuous control-flow blocks (named B_{dst} and B_{dst} respectively), then v_{src} is the last function node in B_{src} and v_{dst} is the first node in B_{dst}.

In the appendix (A.2), we provide an example of a CDG and a CBG to demonstrate how they work together.

5.2.17 Contextual Interprocedural Control Flow Graph (CICFG). The **Contextual Interprocedural Control Flow Graph (CICFG)** is an extension of the ICFG and it describes the complete control flow across all instructions, including *context* [19]. A *context* defines the information needed for an operation to occur. The CICFG is formally defined as $G = (V, E, \xi)$. The nodes $v \in V$ are basic blocks, and the edges $e \in E$ are either intraprocedural control flows, or calling relationships from a node v_{src} to v_{dst}. Lastly, ξ is a set of contexts through which every node $v \in V$ could be reached [19]. The primary difference between the CICFG and the ICFG is that the CICFG gives a more detailed analysis of a program because the context allows to differentiate between different instances or paths that a function may be called. Two examples of a context are *user-aware* and *user-unaware*, which indicates whether the user is aware of what operations or resources an application or piece of code is using. For example, if an app is using the user's location, in a user-aware context, the user knows that the app is using their location, whereas in a user-unaware context, the user would not know about the app using the user's location [26].

5.2.18 Contextual API Dependency Graph. The **Contextual API Dependency Graph** (**CADG**) is built from a CICFG. Not all of the nodes of the CICFG are security-related or invokes a sensitive API. The CADG is an abstraction of the CICFG that only focuses on the security sensitive API invocations [19]. A CADG $g = (V, E, \alpha)$ is a directed labelled graph, with nodes $v \in V$, edges $e \in E$, and labelling function α. The set of nodes V in a CADG represent the *basic blocks* of the program. The edge set E in a CADG represent the *data flow* between the basic blocks. A CADG edge $e_{cadg} = v_{src} \longrightarrow v_{dst}$ indicates that v_{dst} uses data that has been defined by the basic block v_{src}. The labeling function $\alpha : V \longrightarrow \Sigma$ associates nodes with the labels of corresponding contextual API operations. Each label consists of API prototype, entry point and constant parameter [192].

5.2.19 Contract/Semantic Graph. A **Contract Graph** [103] (or a **Semantic Graph** [108]) is a representation created specifically for vulnerability detection in smart contracts. The set of nodes V in a contract/semantic graph is partitioned into three types: *core node* V_{core}, *normal nodes* V_{norm}, and *fallback nodes* V_f. A *core node* $v_{core} \in V_{core}$ represents the key invocations and variables that play a crucial role in detecting vulnerabilities. A *normal node* $v_{norm} \in V_{norm}$ represents invocations and variables that can assist in detecting vulnerabilities, although they do not have the same significance as core nodes. A *fallback node* $v_f \in V_f$ simulates the fallback function that is incurred on a contract attack.

The edge set E in a contract/semantic graph has three partitions: *control flow* edges $E_{cf} \subseteq (V_{core} \times V_{norm}) \cup (V_{norm} \times V_f) \cup (V_f \times V_f)$, *data flow* edges $E_d \subseteq (V_{core} \times V_{norm}) \cup (V_{norm} \times V_f) \cup (V_f \times V_f)$, and *fallback* edges $E_{fall} \subseteq V_f \times V_f$. A *control flow* edge $e_{cf} = v_{src} \longrightarrow v_{dst}$ indicates that v_{src} can flow to v_{dst} in the next step of the program. A data flow edge $e_d = v_{src} \longrightarrow v_{dst}$ indicates that v_{dst} receives data from v_{src}. A fallback edge $e_{fall} = v_{src} \longrightarrow v_{dst}$ indicates that v_{dst} is the fallback node and v_{src} is the call.value invocation, which is the invocation in a smart contract that can cause a reentrancy vulnerability, if the code is vulnerable [108]. It can also mean that v_{dst} is the function under test, and v_{src} is the fallback node. Generally, this edge indicates interactions with the fallback function [46].

In the appendix (A.2), we provide an example of a CADG and a Contract/Semantic Graph for a snippet of source code to demonstrate how the code is abstracted into these graph forms.

5.2.20 Contextual Permission Dependency Graph. The **Contextual Permission Dependency Graph** (**CPDG**) [26] is also built from a Contextual Interprocedural Control Flow Graph (CICFG). The CPDG is an abstraction of the CICFG that only focuses on functionality related to Android permissions [26]. A CPDG $g = (V, E, \lambda_p, \xi)$ is a directed labelled graph, with nodes $v \in V$, and edges $e \in E$. Nodes in a CPDG represent the program's *basic blocks* whose functionality pertains to using Android permissions. Edges in a CPDG represent the *data flow* between the *basic blocks*. A CPDG edge $e = v_{src} \longrightarrow v_{dst}$ indicates that there is a path from v_{src} to v_{dst} in the CICFG, and that both nodes are in the same function. λ_p is the set of labels representing the concerned permissions. ξ is a set of contexts through which every node is the CPDG could be reached [26].

5.2.21 Contextual Source and Sink Dependency Graph. The **Contextual Source and Sink Dependency Graph** (**CSSDG**) [26], is also built from a Contextual Interprocedural Control Flow Graph (CICFG). The CSSDG is an abstraction of the CICFG that considers only the nodes whose functionality is related to using *sources* and *sinks.* [26]. Sources are where sensitive data enters a program and sinks are where they perform security critical operations. This sensitive data flow could be a point of a vulnerability if the data is not handled properly [96]. Thus, a CSSDG $g = (V, E, \lambda_s, \xi)$is a directed labelled graph, with nodes $v \in V$ and edges $e \in E$. Nodes in a CSSDG represent the basic blocks of the program whose functionality is related to using sources and sinks whereas the edges represent the data flow between the basic blocks. λ_s is the set of labels representing the concerned sources and sinks. ξ is a set of contexts through which every node is the CSSDG could be reached [26].

5.2.22 Slice Property Graph. **Slice property graphs** were proposed by Zheng *et al.* [111] and aim to preserve the semantics and structural information that is relevant to vulnerabilities. It also aims to eliminate irrelevant information to reduce the complexity of the graphs. The graph uses SyVCs (Syntax-based Vulnerability Candidates) as slicing criterion to extract the slice nodes that are relevant to vulnerabilities. Then, edges from the Code Property Graph are used as edges between the nodes in the SPG.

5.2.23 Token Graph. **Token graphs** [112] are built from tokens, connecting them via index-focused construction. A token graph $g = (V, E)$ is a directed graph, with nodes $v \in V$ and edges $e \in E$ where $E \subseteq V \times V$. The set of nodes V in a token graph are individual tokens from the source code. For example, a set of nodes can be $if, x, ==, y$. The edge set E in a token graph define a co-occurence relationship between tokens. The co-occurrences describe the relationships between tokens that occur within a fixed-size sliding window [193].

5.2.24 Propagation Chain. A **Propagation Chain** [104] exists when there is a code sequence among a number of specified code snippets. The sequence has direct or indirect data and control dependencies between adjacent code snippets. The propagation chain set $PC(a, b)$ denotes the set of propagation chains between two code snippets a and b. Each program snippet will have propagation chains that affect it and propagation chains that are affected by it. In terms of vulnerability detection, a vulnerable, or defective, propagation chain denotes a code sequence from the vulnerable code to the program's vulnerable output. The set of defect propagation chain, called the defect propagation chain set, is denoted as $EPC(d, f)$ and is a subset of the propagation chain set $PC(d, f)$ from a code snippet d to the program failure code f. Propagation chains can be constructed by data flow or control flow relationships. Zhang *et al.* [104] use data flow relationships to create the propagation chains for smart contracts. In this instance, the data flow graph is defined as a set of nodes and edges $g = (V, E)$, where the nodes $v \in V$ represent variables in a smart contract and the edge set E denote the dependency relationships between them. For example, $e = v_{src} \longrightarrow v_{dst}$ denotes that v_{dst} has a data relationship or dependence to v_{src}.

5.3 Lexical Representations

Lexical representations describe representations that are focused on words and vocabularies. These representations do not show relationships between nodes, as done in graph representations. Lexical representations are also primarily based upon NLP work.

5.3.1 Tokenizer. A **tokenizer**, which can also be referred to as a *lexed* representation, takes source code and creates individual tokens for every word or symbol [63]. This is largely based off of existing NLP techniques. While there are a variety of different tokenization algorithms that can be used (*e.g.*, SentencePiece), a majority of the studied papers did not specify what algorithm was used beyond simply stating 'the code was tokenized'. A few papers, instead, specified the library they used to tokenize the code, *e.g.*, Python's tokenizer [117], javalang tokenizer [40], ANTLR [15], js-tokens [120], Clang [137], and phply [86].

For the papers that explicitly stated their tokenization algorithm, we found papers using **Byte Pair Encoding (BPE) Subword Tokenization** [119, 122], **SentencePiece** [54], character-level tokenizers [39, 54], **WordPiece** [129, 138], white space tokenizers [133], and sentence-level tokenizers [87]. The **Byte Pair Encoding (BPE) Subword Tokenization** is a method that breaks up whole words into smaller parts, in an effort to compress the tokenized data. Frequent words are represented as individual tokens, but infrequent words are split into multiple subword tokens. For example, if the pair of tokens "a" and "b" happen frequently, then they will be combined and become the single token "ab" [194]. **SentencePiece** is another subword tokenization method and employs lossless tokenization, where all the information needed to reproduce the normalized text is preserved in the output of the encoder (*i.e.*, it treats the input text as a sequence of Unicode characters [195]). **WordPiece** tokenizes a word using MaxMatch, a process that involves iteratively picking the longest prefix of the remaining text that matches a vocabulary token until the entire word is segmented [196]. In the appendix (A.3), we provide an example of BPE subword tokenizer, a standard tokenizer, SentencePiece, and WordPiece for the same source code, to demonstrate the differences between the algorithms.

Tokenizers that use strong embedding algorithms such as `word2vec` [57] are able to capture the semantic meaning of the code. When the vector embeddings are created from the tokenizer, these numbers are largely based off the semantic relationship with another word. That is, if a word is semantically related to another, their vector representations will be similar [57]. These techniques can be particularly useful when the model needs to learn the semantics of a chunk of code in order to complete the task at hand. It is also quite simple to tokenize source code, with a complexity of $O(n)$. A built-in function for nearly any language will simply take in a line of text, and break it up into tokens based on a provided delimiter. Models such as `word2vec` and `doc2vec` [57, 197] are very developed and are a great way to create word embeddings from a vocabulary. This is a reason why this representation is also so popular. However, tokenizers do not capture the structural properties of source code, and this representation thus lacks the ability to understand the syntax of a program.

5.3.2 iSeVC and sSyVC. A **sSyVC (source code- and Syntax- based Vulnerability Candidate)** are features of code that have some vulnerability syntax characteristics. An example would be for vulnerabilities that are associated with pointers. A sSyVC would be a line of code which contains a '*' since this symbol is what is used when dealing with pointers [158]. These characteristics are obtained through ASTs. [158]. **iSeVC** stands for **intermediate code and Semantics-based Vulnerability Candidate** and are derived from sSyVC using program slicing. The sSyVC are the nodes of interest, and the PDG of the program allows one to perform the forward and backward slicing, as described in § 5.2.7. The resulting set of ordered statements, all containing data or control dependencies between them, are the iSeVCs [158]. iSeVCs contain information regarding data and control dependence, hence their name which relates them to semantics [48].

5.3.3 Contract Snippet. A **Contract Snippet** [47] contains key program statements or lines from a smart contract which could induce a vulnerability. These contract snippets are aimed to be highly expressive such that more pertinent features can be extracted. The contract snippets are all semantically related by control flow dependence, and all highlight a key element (*call.value*) in reentrancy detection (which the paper this representation is proposed in focuses on). Contract snippets can be generated by control flow analysis. Once the contract snippets are created, they are then tokenized and transformed into feature vectors.

5.4 Miscellaneous Representations

Miscellaneous representations are those that do not fit into any of the previously described categories.

5.4.1 Image. Prior works used the already very developed techniques in image analysis to analyze aspects of code in order to detect vulnerabilities [87, 92] and malware in Android applications [21]. The core idea behind this method is leveraging visual patterns in software to detect anomalies or similarities. This technique allows researchers to take advantage of the techniques developed for detection of elements in regular images.

5.4.2 Code Metrics. **Code metrics** are a quantitative measure that relates certain features to a numerical value, namely the number of times the feature occurs [198]. Code metrics can be defined differently for different tasks. Some common metrics include *lines of code*, *code churn* (*i.e.*, how often code is changed), and more. Prior works also introduce new metrics for a particular purpose, such as SQL injection [156]. Rather than lines of code, or other classical metrics which would not be useful in SQL injection detection, metrics such as number of semicolons, presence of always true conditions and the number of commands per statement provides more relevant information that would result in better predictions for SQL injection. These metrics can be related to a *risk* factor dictating how much of an impact the metric could have on code to create a security issue [198].

5.4.3 Code Gadgets. **Code Gadgets** are essentially a method to describe or represent a program slice. They have a number of ordered code statements or code lines that are semantically related to each other by data or control dependency [41]. Code gadgets were created for the exact purpose of vulnerability detection [41], which can explain the reason for its popularity in security related tasks.

5.4.4 Opcode Sequences. An **opcode**, or operation code, specifies the operation to be completed for an instruction [199]. They, in particular, specify the lowest-level operation to be completed such as `PUSH`, `MSTORE`, and `CALLVALUE` [164]. These features can be used to understand on a low level what the code is doing. Opcodes are learned as vectors, and Liao *et al.* [164] using n-grams and `word2vec`[57] to learn them as embeddings. Since opcodes already dictate operations in computers, this representation is simple to generate.

5.4.5 Regular Expression. A **regular expression** is a sequence of characters that defines a search pattern, often used for string matching or manipulating text within strings based on specified rules. One paper represented source code as regular expressions, which encode information about the API calls in the code. Specifically, the work described by Liu *et al.* [180] first takes an Android application and creates a CFG from the callbacks. The CFG is transformed to an ICFG, reduced to an API graph, and then an automata to regular expression algorithm is used to generate the regular expressions. This solution is used for multifamily malware classification and addresses the issues of recognizing malware family behavior patterns, code obfuscation and polymorphic variants that are commonly used by attackers to evade detection. The regular expressions describe the behavior patterns of malware families. While this method can be computationally expensive, as three graphs have to be made before being transformed to a regular expression, it allows to capture the differences between malware families.

5.4.6 Application Information. While this representation can be sorted under Code Metrics, the features extracted by [23] more accurately fall under the name of **Application Information**. In this representation, an Android application is reverse engineered to extract the original Java files and Android XML. From these files, the API Calls made, and the permissions used are extracted. Other features such as *is crypto code* or *is database* that specifies certain features of the code that might be associated with malware [23] are also extracted. This representation is a method of feature extraction. Once these features are extracted, they are transformed into vectors of features, and given a label as being benign or malicious. Then, this data can be used in an ML classifier.

5.4.7 API Calls. Wang *et al.* [24] uses **API Calls** extracted from the source code of an Android application, along with permissions extracted from the Android Manifest file. This paper uses a tool called DroidAPIMiner to extract the top 20 API calls that are called by malicious applications. Using these API Calls and permissions as features used to train a deep learning model allows [24] to find malware in Android applications. Once again, this representation is a method of feature extraction, and the features are then converted into vectors which are then passed to a model.

RQ1 Findings:

- There are 39 representations offering a variety of information about the source code, although a common goal is to capture the semantic and syntactic information in code.
- There are 24 unique graph-based representations, the most popular source code representation category, as it shows the relationships between different nodes (*e.g.*, lines or statements) and how they interact.
- Although there are a larger variety of graph-based representations, 47 papers used tokenizers and 32 papers used ASTs as their representations.
- Seven papers [43–48, 101] propose a representation unique to their application (*e.g.*, a contract graph [46] to find smart contract vulnerabilities). Such task-/language-specific representations are not generalizable to other languages and/or purposes.

6 RQ2: DO CERTAIN TASKS ONLY USE OR MOSTLY USE ONE TYPE OF REPRESENTATION?

Figure 2 depicts the relationships between *representations* and *cybersecurity tasks*. **ASTs** and **tokenizers** were the two representations most commonly used for **vulnerability detection**. Given how popular LLMs and NLP have become in recent years, it follows that a tokenizer would be a popular representation, as a tokenizer is what is used for NLP techniques. The vast availability of pre-trained NLP models gives developers the power to easily fine-tune these models on whatever task they desire, without great cost [200]. ASTs are also a relatively cost-effective representation, as compilers use ASTs to represent the structure of source code. This also allows developers to generate this representation with relative ease, while also providing crucial structural details about source code that can expose vulnerabilities [175].

Although AST and tokenizers were the most used representations, we also found that vulnerability detection techniques mostly used **graph-based representations**. Graph-based representations were also popular for other cybersecurity tasks, namely **malicious code localization**, **malware detection**, **vulnerability localization**, **vulnerability analysis**, **malware classification**, and **vulnerability extrapolation**. Among the graph-based representations, **CFG** is the most popular one. This could be because it details the program's execution flow, which can be useful in finding whether a vulnerability would occur due to the structure of the program. A developer may choose a CFG over another representation, such as an AST, because it provides more detailed information about the source code. For example, compared to a call graph of a program from the dataset used in [35], the CFG has 150,000 vertices/nodes, whereas the call graph has 10,000 nodes. While being more computationally expensive than some other methods (such as a tokenizer or AST), the information provided by the graph allows for a more robust understanding of code.

DFG is another popular representation, particularly for **vulnerability detection**. Similarly to **CFGs**, **DFGs** detail the flow of the program, although they detail the *data flow*. This can also be useful in vulnerability detection

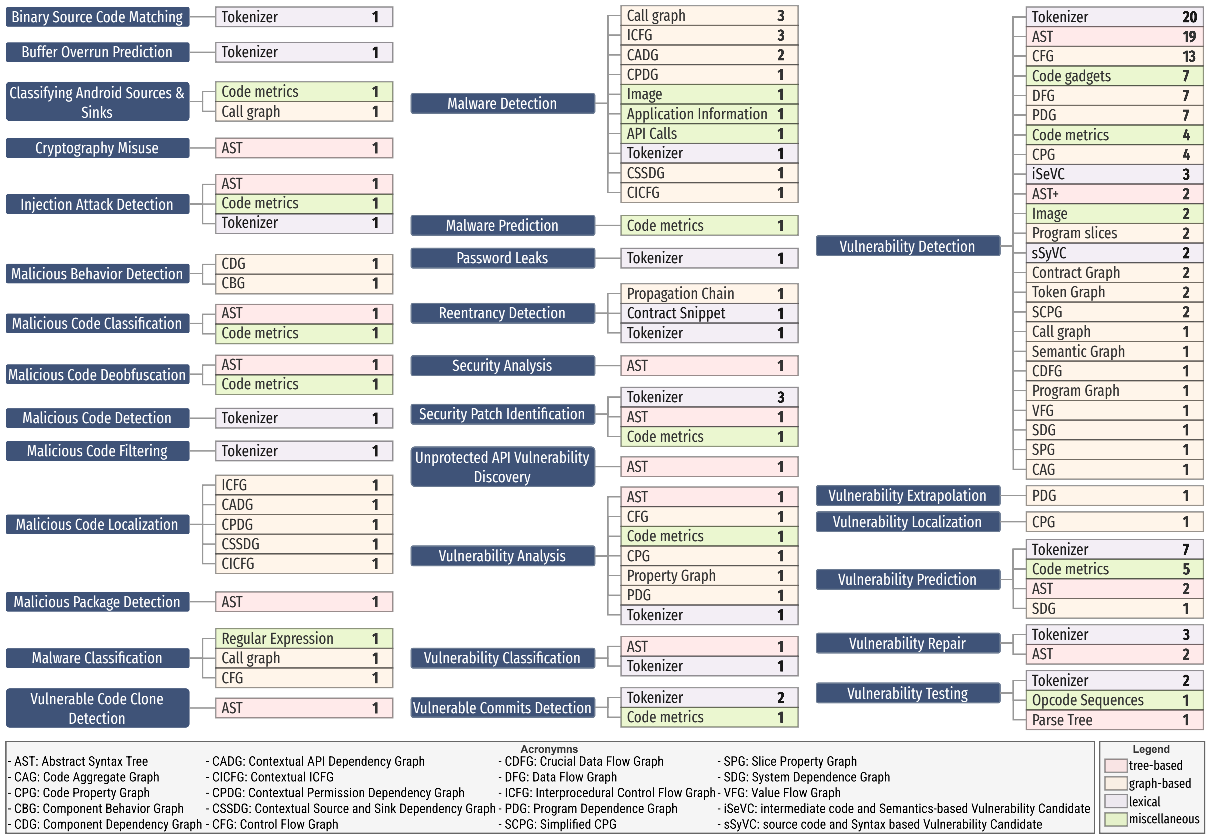


Fig. 2. Relationship between representations and tasks

because if a harmful data input reaches a security sensitive program point, then a vulnerability will likely occur. Hence, one may choose a DFG over a CFG for detecting a specific type of vulnerability related to data flows. Graph-based representations are useful for these types of tasks because they detail the overall structure and flow of code. In order to find *sinks*, or points in code where a dangerous function call is invoked, it is helpful to understand the structure because one can pinpoint what function call starts or invokes the vulnerability. Additionally, depending on the type of information shared within the graph, one can better pinpoint areas of code which contains a vulnerability.

Code gadgets, **iSeVC**, and **sSyVC** were representations that were specifically made for **vulnerability detection** techniques. These representations are focused particularly on vulnerability candidates or potentially vulnerable snippets of code. This zoomed-in view allows for a more targeted approach towards vulnerability detection. One may choose this representation for a more light-weight approach to the problem, compared to a more resource intensive method such as a **CPG**.

ICFGs, **call graphs**, and **CADGs** are the preferred representations for **malware detection**. **ICFGs** describe the complete control flow across a program, and **CADGs** are derived from a version of **ICFGs** that include context. Both of these representations provide insights to potential security related invocations, which could allow malware to affect the system. Understanding what calls are made and how instructions flow in a program can give insight to any irregular or unsafe operations done by the code. ICFGs are known to be robust against evasion and obfuscation methods that are used by malware [18]. This could be due to the fact that the ICFG gives the view of the whole program, and malware typically interacts with an entire system beyond just a singular procedure. Rather, malware may call functions outside of a procedure, and the ICFG would give insight to these calls, as well as any unusual or unexpected activity [26].

When choosing a representation for a particular task, it is important to consider the needs and resources of the developer for the application. For example, a **CPG** is a highly robust graph that is complex to generate. If the developer *needs* a highly robust graph for their task and has the *resources* to develop the CPG (for thousands of samples of code), then a CPG would be a good option as a representation. However, if the developer does not have the time or resources to have such a robust graph, but knows that the issue they are trying to detect pertains to the way data flows through a program (*e.g.*, detecting injection attacks), then they might want to choose a **DFG**, as it would provide the information needed without being overly complex or resource intensive.

RQ2 Findings:

- Since vulnerability detection is the most popular task, it is the task that most representations are used for.
- Certain representations (*i.e.*, **iSeVC**, **sSyVC**, and **code gadgets**) were created for the specific task of vulnerability detection and are therefore only used for that task.
- **Call graphs**, **ICFGs** and **CADGs** are the preferred representation for malware detection.

7 RQ3: WHAT CYBERSECURITY TASKS ARE COVERED BY THE ML-BASED TECHNIQUES?

To better understand the types of tasks that are covered by the techniques discussed in this paper, we sorted the different tasks to fit into the 9 disciplines of the Rational Unified Process (RUP) cycle [201]. The RUP cycle is a software development process framework that allows software developers to better organize and plan the development process. In this question, we sorted the unique cybersecurity tasks found from our search into the nine main workflows of the RUP cycle: *business modeling*, *requirements*, *analysis and design*, *implementation*, *test*, *deployment*, *configuration and change management*, *project management*, and *environment*. We observed that the cybersecurity tasks only fit into **5** out of the **9** categories. Those categories being: analysis and design, configuration and change management, environment, implementation, and test. Figure 3 depicts how the different cybersecurity tasks fit into these 5 disciplines from the RUP cycle.

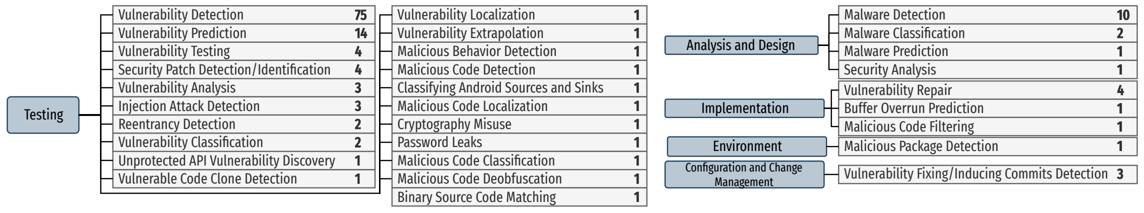

Testing
Vulnerability Detection 75
Vulnerability Prediction 14
Vulnerability Testing 4
Security Patch Detection/Identification 4
Vulnerability Analysis 3
Injection Attack Detection 3
Reentrancy Detection 2
Vulnerability Classification 2
Unprotected API Vulnerability Discovery 1
Vulnerable Code Clone Detection 1
Vulnerability Localization 1
Vulnerability Extrapolation 1
Malicious Behavior Detection 1
Malicious Code Detection 1
Classifying Android Sources and Sinks 1
Malicious Code Localization 1
Cryptography Misuse 1
Password Leaks 1
Malicious Code Classification 1
Malicious Code Deobfuscation 1
Binary Source Code Matching 1
Analysis and Design
Malware Detection 10
Malware Classification 2
Malware Prediction 1
Security Analysis 1
Implementation
Vulnerability Repair 4
Buffer Overrun Prediction 1
Malicious Code Filtering 1
Environment
Malicious Package Detection 1
Configuration and Change Management
Vulnerability Fixing/Inducing Commits Detection 3

Fig. 3. Cybersecurity Tasks in the RUP cycle

7.1 Analysis and design

Analysis and Design involves translating requirements into a formal model of the software, resulting in a system description that guides implementation. We categorize these tasks under analysis and design, as these tasks are analyzing the source code without necessarily implementing a new design. The tasks in this category ensure that, once developers are able to start implementing the system, there are no security issues. **Malware detection, prediction, and classification** are important tasks, particularly for Android applications [17–26, 35, 150, 180], but also for any system vulnerable to malware. Malware allows attackers to take advantage of security flaws in systems. Understanding types of malware and ransomware, as well as where they occur, allows one to take steps to avoid these issues before implementing a system. **Security analysis** is a task that was used particularly for smart contracts [175], but it can be generalized to any system that has the potential to be compromised.

7.2 Implementation

Implementation is the actual coding of the software model. This is when the design from the previous phase is brought to life. **Vulnerability repair** is a task that fixes vulnerabilities in code. This fits into implementation because it is actually implementing the fix to the code while the developer is working [30, 43, 115]. Moreover, there are a few tasks involved with inferring or predicting elements of code, such as **buffer overrun prediction** and **vulnerability prediction**. Buffer overrun is a type of vulnerability, and Choi *et al.* [128] focus their prediction approach on this one type of vulnerability, rather than many others. These prediction techniques allow developers to know where these issues may occur in the code while they are implementing it. We consider these tasks as the implementation phase because vulnerability repair involves implementing a fix to the problem. The prediction tasks, on the other hand, are also considered as implementation because it involves being proactive against these vulnerabilities or issues and finding them before they can be exploited.

7.3 Testing

Testing involves exercising the software for any flaws or design inconsistencies, helping prevent costly security issues during production. Detection tasks indicate that the vulnerabilities already exist in the implemented system. Thus, the systems are being *tested* for these existing issues so they can be resolved. Many detection techniques fall into this category, such as **cryptography misuse, malicious code detection, filtering, classification, deobfuscation, and localization, malicious behaviour detection, password leaks**, **injection attack detection, reentrancy detection**, **vulnerability analysis, detection, classification, localization, extrapolation, and testing**, **vulnerable code clone detection** and **unprotected API vulnerability discovery**. **Binary source code matching** is also used for tasks like malware detection and vulnerability assessment [202].

7.4 Configuration and Change Management.

Configuration and change management tracks and maintains a project as it is evolving through time. It ensures that the code created during implementation is still useable and can be reused throughout other portions of the project if needed [201]. Modern software development workflows often use remote repositories to track the code that has been created and its changes. Thus, commits are usually the source of a vulnerability or issue, and the fix to a vulnerability or issue. Zhou *et al.* [131] and Nguyen-Truong *et al.* [130] created techniques to assist in the **classification of security commits**, as well as **identifying vulnerability inducing, or vulnerability fixing commits**, including **security patch detection**. Due to the nature of these tasks (*e.g.*, specifically classifying, identifying, or detecting changes relating to security), we categorize these tasks under configuration and change management as they are directly related to how the code is changed and reused throughout a project.

7.5 Environment

Environment focuses on the software development environment required for the engineers to develop the system. This includes techniques and processes required by developers [201]. Packages are an essential part of the development process. They provide useful techniques that can greatly simplify the implementation of a system. However, some packages can contain some type of vulnerability or malware that can jeopardize the integrity of the system. Therefore, **malicious package detection** is an important task that can protect systems from such malicious software. We categorize this task under environment because packages are what developers use in their environment as part of their development process. While malicious packages might not necessarily impact their environment directly, it still involves the space in which developers work.

As cryptocurrency becomes a more popular and prevalent topic, research is also starting to focus on creating ML-based techniques to assist in issues relating to cryptocurrencies and smart contracts [46, 47, 79, 101, 103, 108, 164, 166, 175, 176]. All of these papers are interested in vulnerability detection or testing, and security analysis of smart contracts. Two papers [47, 104] focus on a particular vulnerability called *reentrancy attacks*, which is a vulnerability specific to smart contracts. This vulnerability results in an attacker being able to withdraw funds from a smart contract repeatedly and transfer them [47].

While most of the tasks are distinct, some tasks are very closely related to one another. For example, **malicious behaviour detection** is looking for suspicious activities and behaviours that might indicate an attack, regardless of if the attack is using malware, whereas **malware detection** is typically looking for specific features or signatures that are known to be associated with malware [17]. **Detection** tasks focus on generally identifying a certain feature (*e.g.*, **vulnerability detection** focuses on broadly finding a segment of code that is vulnerable), whereas **localization** tasks are focused on pinpointing exactly where, in a larger code base, the feature occurs (*e.g.*, **vulnerability localization** is focused on pinpointing the specific line in which the vulnerability occurs). Similarly, **prediction** techniques are interested in finding issues before they are apparent in a system (*e.g.*, **vulnerability prediction** focuses on finding vulnerabilities before they are launched officially into a system, to prevent an attacker having the opportunity to discover it when it is launched), whereas **detection** techniques are interested in identifying issues that already exist in a working system (*e.g.*, **vulnerability detection** focuses on finding vulnerabilities that are already present in a launched system). **Vulnerability testing**, on the other hand, focuses on evaluating a system to **Cryptography misuse** describes a detection task, where improper uses of cryptography APIs are discovered. Cryptography APIs tend to be hard to use, so this task alerts developers to when they may have made a mistake in their implementation [33]. **Vulnerability extrapolation** describes a task which aims to find potential vulnerabilities based on existing vulnerabilities. This allows developers to find vulnerabilities that may not exist or be known yet based on past vulnerabilities and their structure. This task in particular focuses on how vulnerabilities can spread or appear in similar contexts based on the vulnerability characteristics [91]. **Unprotected API Vulnerability Discovery** refers to discovering instances which a vulnerability can occur when an unprotected API is called or used in a system [174].

RQ3 Findings:
- Vulnerability detection by far is the most popular task, with **75** papers focusing on this task.
- Certain papers focus solely on detecting one type of vulnerability, such as *buffer overrun prediction*, *reentrancy detection*, and *injection attack detection*.
- A majority of these tasks fit under the *testing* category of the RUP cycle, meaning that these techniques are aimed for evaluating the security of already written code before it is deployed.

8 RQ4 WHAT PROGRAMMING LANGUAGES ARE PREDOMINANTLY TARGETED BY THE ML-BASED TECHNIQUES FOR CYBERSECURITY TASKS?

Table 3 shows the programming languages that are covered by existing techniques. Two works [35, 125] did not specify the language(s) used. Papers without language or dataset details are labeled as “not specified”.

We observed that `C` is the most popular language; **81** (**57.4%**) papers developed techniques for `C` programs. `C++` is the second most common language, covered by **50** papers (**35.5%**), all of which also supported `C`. The popularity of the C and C++ languages could be attributed to two main factors: the availability of datasets for cybersecurity tasks [203–206], and the higher risk of memory-related vulnerabilities in these languages [207].

We also observed that a number of techniques focus on security tasks for Android applications [12, 12, 17–27, 35, 96, 150, 180]. Although Android applications can be written using Kotlin and Java, the papers studied in this survey only focused on apps written in Java. Additionally, with the increasing popularity of smart contracts, a number of papers developed techniques for Solidity, the language that smart contracts are written in.

Table 3. Languages supported by existing techniques. JS = JavaScript; TS = TypeScript; SpiderMonkey = SM

Lang.	#Papers	Lang.	#Papers	Lang.	#Papers	Lang.	#Papers	Lang.	#Papers	Lang.	#Papers	Lang.	#Papers
C	81 (57.4%)	**JS**	12 (8.5%)	**Python**	6 (4.3%)	**C#**	2 (1.4%)	**Gecko**	1 (0.7%)	**Powershell**	1 (0.7%)	**SM**	1 (0.7%)
C++	50 (35.5%)	**Solidity**	12 (8.5%)	**CSS**	3 (2.1%)	**SQL**	2 (1.4%)	**Go**	1 (0.7%)	**Ruby**	1 (0.7%)	**XML**	1 (0.7%)
Java	36 (25.5%)	**PHP**	8 (5.7%)	**Rust**	3 (2.1%)	**TS**	2 (1.4%)	**HTML**	1 (0.7%)	**Smali**	1 (0.7%)	**XUL**	1 (0.7%)

We also observed that there are only 6 techniques for Python and 12 for JavaScriptwhich is surprising given their increasing popularity with developers [6]. If they are used so much in practice, we could expect there to be an according number of techniques that aim to cover them, particularly for security-related assistance. It is also possible that there are not enough datasets for these languages and because of this, there are not many techniques for them because researchers are unable to have the necessary data to train and test the models. Nonetheless, this is a gap that should be addressed by future works.

No techniques were language-agnostic, meaning that all the systems created were made only for one or potentially a few languages. Making a tool language-agnostic is difficult due to the variety of programming paradigms that exist, and that all languages do not follow one particular paradigm. This does mean, however, that researchers should be diligent in developing techniques that support popular and commonly used languages.

RQ4 Findings:
- C is the most common language that is targeted by ML-based cybersecurity techniques.
- Despite their popularity, there are not many ML-based techniques for Python and Javascript.
- A large portion of papers are aimed at solving security issues in Android applications. Thus, Java is also a popular language, with 36 techniques geared towards solving these issues in Java.
- Given the increasing popularity of smart contracts, there are a number of techniques (12) that were created for Solidity- the language used to create smart contracts.

9 RQ5: WHAT MODELS ARE COMMONLY USED WITH DIFFERENT REPRESENTATIONS?

To sort the different types of models used throughout the papers, we took inspiration from Siow *et al.* [208] and classified them into five categories: *sequence-based* models, *feature-based* models, *tree-based* models, *graph-based* models, and *neural networks*. While there is overlap between our *neural network* categories and others, our classification is based on the type of inputs that the models accept. For example, a Graph Neural Network may have a neural network architecture, however it is designed to handle graph-based inputs. Table 4 demonstrates the models used by the surveyed papers and how they fall into each category. By far, a majority of these models are *sequence-based*, particularly CNNs, Transformers, and LSTMs. This is most likely due to their general popularity, but also because these models are very powerful, are able to overcome the vanishing gradient problem, and are able to handle long-term dependencies [209].

[6]https://spectrum.ieee.org/the-top-programming-languages-2023

Table 4. Different models used by the surveyed papers.

Model Type	Models
Sequence Based	Transformer [43, 44, 54, 90, 99, 104, 115, 119, 122, 144], Discrete Fourier Transform [51], Convolutional Neural Network[11, 21, 36, 38, 87, 92, 126, 132, 139, 151, 161, 178], Deep Pyramid Convolutional Neural Network [145], Text CNN [39], Temporal CNN [14], Recurrent Neural Network [38, 40, 90, 137, 158], Bidirectional RNN [127, 173], Autoencoders [30], seq2seq [84, 144], BERT [135, 162], codeBERT [129, 138], JavaBERT [138], Hierarchical Attention Network [89, 95, 172], doc2vec [141], word2vec [120, 141], Self attention networks [95], Gated Recurrent Unit[174], Bidirectional Gated Recurrent Unit [86, 160, 171, 178], Online Learning [19], Long Short Term Memory [11, 117, 123, 134, 135, 152, 159, 163, 165], Bidirectional LSTM [13, 16, 41, 47, 95, 110, 135, 160, 169, 178], Paragraph Vector Distributed Memory [166], Deep Learning Attention-based Convolutional Gated Recurrent Unit [124], Bidirectional RNN for Vulnerability Detection and Locating [48], Passive Aggressive Classifier [18], Extreme Learning Machine [150], Encoder-Decoder [142], GPT [143]
Graph Based	Graph Neural Network [12, 45, 46, 78, 80, 93, 98, 102, 105, 107, 108, 112–114, 118], Deep Belief Network [24], Graph Convolutional Network [35, 81, 86, 95, 113, 177], Graph Attention Network [97], Feature Attention- Graph Convolutional Network [88], Recurrent Graph Convolutional Network [111], Gated Graph Neural Network [85, 99], Bidirectional Graph Neural Network[37], GraphCodeBERT [101]
Tree Based	Boosting [155, 156], Extreme Gradient Boosting [125], Light Gradient Boosting Machine [125], adaboost[91], Gradient Boosting Decision Tree [91], Bagging [156], Random Forest [20, 22, 23, 25, 27, 53, 82, 83, 91, 109, 125, 146, 148, 149, 155, 163, 164, 175], coForest [109], Decision Tree [23, 27, 83, 91, 146, 148, 153, 155, 163, 164, 175], code2vec [168], Naive Bayes [17, 23, 27, 53, 83, 125, 148, 153, 163], Tree augmented Naive Bayes [83], Abstract Syntax Tree Neural Network [34], Extra-trees Classifier [36]
Neural Networks	Multi-Layer Perceptron [22, 79, 82, 91, 94, 107, 114, 136, 152, 170], Neural Network [103, 163, 175, 180], Deep Neural Network [11, 106, 146, 157, 167], Complex Deep Neural Network [146], Abstract Syntax Tree Neural Network [34], Attention Neural Network [100], Neural Memory Network [128], Random Neural Network [132]
Feature-Based	Clustering [91], K-means clustering[141], K-median clustering [179], Naive Bayes [17, 23, 27, 53, 83, 125, 148, 153, 163], Gaussian Naive Bayes, Tree augmented Naive Bayes [83], logistic regression [17, 82, 83, 91, 94, 109, 125, 146, 148, 153, 154, 164], Extreme Machine Learning [150], Nearest Neighbor [20], K-Nearest Neighbor [22, 91, 146, 153, 163, 164], linear regression [146], Support Vector Machine [20, 23, 26, 27, 33, 91, 96, 116, 125, 133, 140, 141, 146, 147, 154–156, 163, 164, 175], C-Support Vector Classification Variant of Support Vector Machine[146], Linear Discriminant Analysis [91, 156, 163], Density-Based Spatial Clustering of Applications with Noise [141]

However, SVMs overall were the most popular model. This might be because when determining if code has some sort of cybersecurity issue, such as a vulnerability, the most useful thing to learn or focus on are the features of the code. For example, if a model can learn what features make a piece of code vulnerable or not vulnerable, then the model will be able to successfully perform vulnerability detection. SVMs are powerful in learning the features that differentiate classes, or in this case, code. Therefore, SVMs are useful for learning the features of code that would make it vulnerable or malicious versus benign.

Figure 4 shows the types of models that were used for each type of representation. Additionally, the red boxes indicate model types that could potentially be used by these representations, although we did not observe it explicitly in our data. We consider a representation as potentially usable by a model if there are no further transformations needed to get the data into the form the model needs. In other words, this categorization is based on what representations are readily accepted by models. Many papers also used multiple different models to compare the performance of their method. Naturally, there is a constraint on the type of representation and the type of model used. For example, only graph representations can be used with a graph-based model.

RQ5 Findings:

- Sequence-based is the most common and popular category of models.

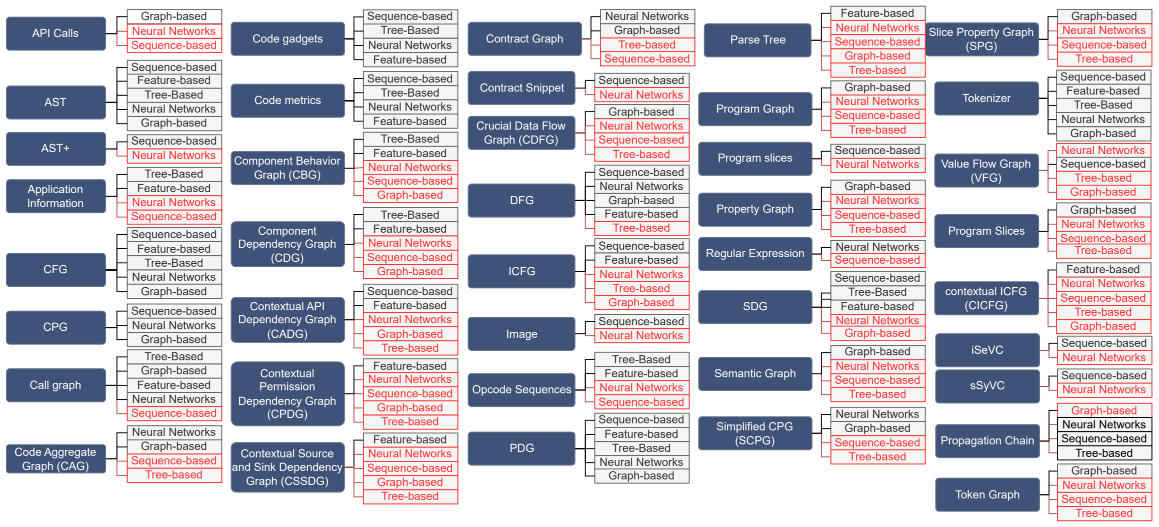


Fig. 4. Relationships between models and representations

- SVMs overall are the most popular model used for different tasks due to its ability to discriminate features that would group code into one group or another.
- The studied papers have a heavier focus on adjusting the models rather than investigating the representation of source code on the whole.

10 THREATS TO VALIDITY

In this section, we will discuss the threats to validity of this survey around construct, internal, and external validity threats as outlined by Runeson and Höst [210].

Construct validity refers to how well the operational measures that are used represent the research questions that are being studied. In our study, these measures mainly involve counts of tasks and representations, as well as the relationship between the two, and the relationships between representations and models. Thus, our analysis relies on the accuracy of the the reviewers while we were categorizing each of the papers. Additionally, when searching for the papers, we relied on the ability of the search engines we used to return to us all the papers that were related to our search query. To mitigate these issues, we had the reviewers separately analyze each paper, and later discuss and resolve any discrepancies in the analyses. We also created a thorough and broad enough query to ensure the results we received from our search encompassed all the papers related to this study.

Internal validity describes how well a study mitigates bias and systematic error such that a causal conclusion can be drawn. Due to the potential threat of incorrectly categorizing a paper, we had two of the authors individually review the papers, and then meet to discuss any discrepancies between the categorizations. Disagreements were resolved through discussion. To assess the reliability of our evaluation, we used Cohen's Kappa. Our calculated score is **0.97**, meaning that we had a near perfect agreement in our analysis [211].

External validity assesses the generalizability of the study. The main threat to this work is that we only focus on the past ten and a half years (2012- May 2023), so we may have missed papers outside of this range. We also did not include preprints and thus might have missed newer papers. Additionally, our keywords rely on machine learning, and any papers that do not include these keywords may have been missed. Finally, we only looked at three specific sources: ACM, IEEE, and Springer Link. While there may have been papers outside of these three sources relevant to our work, they were outside of the scope of search for this paper.

11 FINAL CONSIDERATIONS

In this section, we discuss our findings, share recommendations for future works, and conclude the work.

11.1 Discussion

Aside from our five RQs answered in this SLR, we also identified key findings from our comprehensive analysis related to source code representation:

- **Graph complexity provides more information:** Many papers found that using a graph representation allows to capture more information about the source code, resulting in better performance [37, 41, 44, 78, 81, 86, 89, 93, 98, 105, 106, 110, 113, 114, 135, 158, 161]. Thoroughly capturing the semantic and syntactic relationships and dependencies showed to be crucial for certain tasks, particularly vulnerability detection [41, 44, 78, 81, 86, 89, 93, 98, 105, 106, 110, 113, 114, 135, 158, 161]. Additionally, [113] found that using a graph representation allows to reveal hidden connections within the source code, which was found to be useful in vulnerability detection and addressed limitations of traditional methods.
- **Capturing structure/syntactic information is important:** Another common trend is that the structure and syntax of source code is an important aspect which impacts a model's ability to fully learn source code [19, 44, 78, 85, 86, 93, 98, 105, 106, 114, 136, 152, 158, 161, 164, 175]. One study found that in general, graph representations perform better, particularly for vulnerability detection, because the meaningful information relating to structure, syntax, and semantics is preserved through graph representation [98]. In general, one study found that vulnerabilities are related to code structure and patterns [175]. Additionally, other studies found that including some sort of information regarding the structure of the code on top of a representation which does not consider the structure helped to improve the overall performance [85, 86].
- **Extracted features impact performance:** While seemingly self explanatory, the features one extracts about the source code will impact the performance of the model because it determines what the model is able to learn. In the case of vulnerability detection, the extracted features may not be associated with some vulnerabilities, thus rendering those features, for lack of a better term, useless to the model [15, 105, 132, 154, 164, 175]. Similarly, certain representations offer more information about the source code than others (*i.e.*, a Code Property Graph versus an Abstract Syntax Tree) which enhances the expressiveness of semantics, resulting in more informative extracted features, thus leading the model to perform better [31, 37, 41, 45, 110, 119, 158].
- **Considering code as text does not capture behaviours well:** Many studies agree that treating code as plain text does not provide enough information for a model to learn about the code [43, 44, 81, 89, 105, 119, 132, 135, 136, 158, 164]. Code has more semantic structures that differ from natural language [89], and representing code as plain text causes a loss in this semantic information, such as execution logic and relationships between the code, which is crucial for tasks such as vulnerability detection [44, 81, 105, 132]. One paper found that using BPE subword-level tokenization is more useful than word-level tokenizing [119], as a result of the tendency for code to contain uncommon keywords rather than natural language. Using word-level tokenization, as is done in Natural Language Processing, results in vector representations that are not particularly meaningful, resulting in poor performance [43, 119].
- **Capturing context improves performance:** Including contextual information helps models better distinguish code features, leading to improved performance [19, 26, 100, 172]. In particular, this contextual

information can decrease the number of false positives. In the example of vulnerability detection, added context can assist in precisely distinguishing malicious neighborhoods from benign ones [19].

- **In certain instances, tokenizers can be useful:** While a majority of papers state that tokenizers are not optimal for cybersecurity related software engineering tasks, two papers point out that there are instances in which tokenizers might be a beneficial representation [15, 122]. Tokenization proves to be useful in making certain syntactic and semantic information easily available to simple neural architectures, which use it for fine tuning on a downstream task [122]. Additionally, one paper found that improving the dictionary by increasing its size and complexity when tokenizing also improved the performance of the method [15]. As previously mentioned, BPE subword-level tokenization is shown to improve performance [43, 119], thus suggesting that while a standard tokenizer (such as one used in an NLP model) may not be the most useful representations, there is room to improve tokenization methods such that more pertinent information is passed to the model.

11.2 Recommendations

From the **141** papers we have studied, we noticed that rather than attempting to find a new or more comprehensive way to represent source code, prior works are focused on trying different or new models with more advanced architectures so that the model can learn more from the representation, without changing it. Given the improvements in power and capability of ML models in recent years, it is understandable why researchers would take this approach. However, it is important to remember that the way a model learns is impacted greatly by the feature representations, as it can allow the model to learn and isolate critical information that is important for it to successfully complete its task.

Based on the results presented in Sections 5– 9, we recommend future works to explore different representations for a particular task rather than just fine-tuning the model. Since the model learns features of the source code from the representation, there would be a greater improvement in performance if more attention is paid to the representations. It is also important for researchers to pay attention to the popularity of languages, as there should be an effort to create techniques and tools that address popular and more frequently used languages to ensure that we can avoid as many security risks as possible.

When choosing a representation for a particular task, one might consider the capabilities of different representations. Table 5 highlights these abilities based on key features. ‘Lightweight’ refers to how complex the representation is to create. For example, as discussed above, ASTs are simple to generate. Because their computational complexity is relatively low, they can be considered lightweight. Program slices, on the other hand, have a lot of overhead and thus are not considered lightweight.

Additionally, one may also want to consider the performance of these representations for particular tasks compared to one another. Figure 5 shows the comparison of the average Precision, Recall, and F1 scores for certain tasks, as reported by the studied papers. For space considerations, we only demonstrate in this paper the precision, f1 and recall scores, and only for tasks which had multiple representations. The rest of the metrics data in full is available on our GitHub repository for the project.

We can see from Figure 5 that for vulnerability detection, the Code Property Graph (CPG) has the best recall (99.0%), program slices has the best precision (94.8%), and the program graph has the best f1 score (94.5%). Interestingly, all of these representations are graph based, demonstrating the importance of structural information when detecting vulnerabilities. As a CPG is an incredibly robust graph, giving insights to many aspects of a program, it follows that this representation would perform incredibly well. One thing to consider is that because it is such a robust representation, it is complex to generate, which may not be ideal for someone who does not have the resources needed to generate such a graph. Broadly speaking, graph representations show strong

Table 5. Abilities the different representations in this study have. Representations that have the same abilities are grouped together. BPE is BPE Subword Tokenization, Op. Seq. is Opcode Sequences, and App. Information is Application Information

	Structure	Semantics	Data Flow	Statement Flow	Lightweight	Interprocedural
CPG, PDG, Program Slices, SDG, SPG, CAG	✓	✓	✓	✓	✗	✓
VFG, Contract/Semantic Graph Program Graph, Propagation Chain	✓	✓	✓	✓	✗	✗
CADG, Property Graph, CPDG	✓	✗	✓	✓	✗	✓
CICFG, Call Graph	✓	✗	✗	✓	✗	✓
SCPG, CSSDG	✓	✗	✓	✓	✗	✗
AST, Parse Tree, Image	✓	✗	✗	✗	✓	✗
Tokenizer, Opcode Sequences	✗	✓	✗	✗	✓	✗
iSeVC, Code Gadgets	✓	✓	✓	✓	✓	✗
Contract Snippet	✗	✓	✗	✓	✓	✗
App. Information, API Calls, Code Metrics	✗	✗	✗	✗	✓	✗
CFG	✓	✗	✗	✓	✗	✗
DFG	✓	✓	✓	✗	✗	✗
ICFG	✓	✗	✗	✓	✗	✓
CDG	✓	✓	✗	✗	✗	✗
CBG	✓	✓	✗	✓	✗	✗
CDFG	✓	✗	✓	✗	✗	✗
Token Graph	✗	✓	✗	✗	✗	✗
sSyVC	✓	✓	✗	✗	✓	✗
Regular Expression	✓	✗	✗	✓	✓	✗

results across a variety of tasks, thus speaking to their ability to convey the crucial information relating to the source code. However, in vulnerability prediction and analysis, a tokenized representation did have the best performance compared to other methods (80.1% recall and 96.8% recall, respectively). Code metrics also showed strong results for security patch detection/identification (91.4% f1 score), as well as vulnerability fixing/inducing commits detection (92.0% precision), and vulnerability analysis (96.8% recall). Code metrics are a very lightweight representation, and its performance suggests that the features it presents (such as lines of code or code churn) provide enough information about the source code to perform well in tasks relating to security related code changes and vulnerability analysis. Siow *et al.* [208] found similar results to us in their empirical study on code representations for code intelligence tasks such as code classification. The paper found that graph-based representations are the best at representing program semantics, thus resulting in the best performance compared to tree-based, feature-based, and sequence-based representations. Additionally, the paper found that different tasks require task-specific semantics to achieve the best performance, but on the whole a composite graph that represents program semantics comprehensively will produce strong results.

11.3 Conclusion

In this manuscript, we studied 141 papers to examine the state-of-the-art of source code representations in ML models used for cybersecurity-related tasks. We found that ASTs and a tokenized representation are the most common way to represent source code. We also found that vulnerability detection, malware detection and vulnerability prediction are the most covered tasks by existing techniques. Additionally, we observed that C was the language covered by the most techniques, followed by C++.

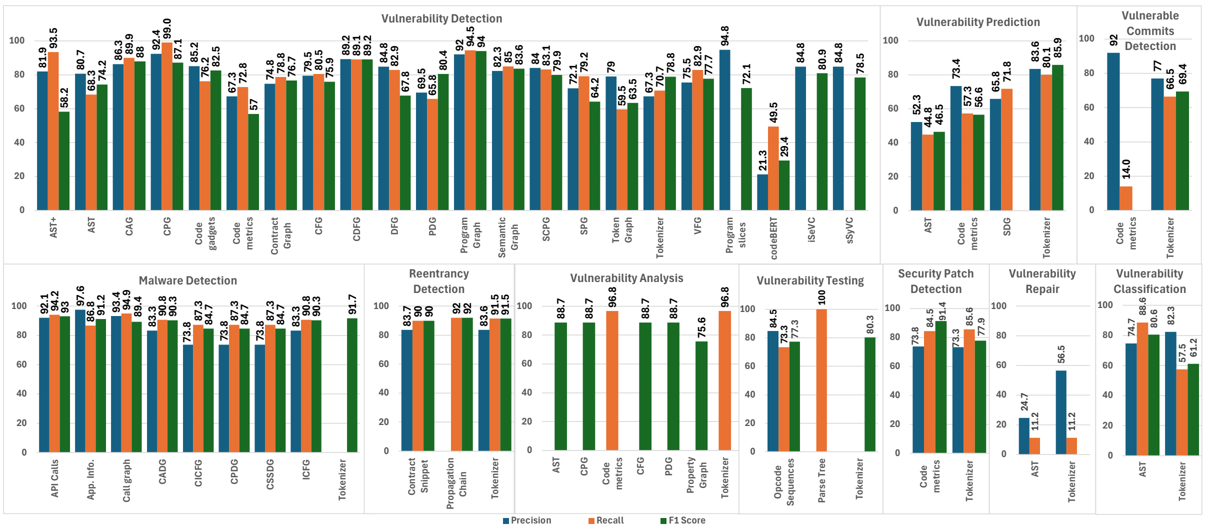


Fig. 5. Aggregated metrics for each representation by task. The abbreviations are as follows: App Info = Application Information; CADG = Contextual API Dependency Graph; CICFG = Contextual ICFG; CPDG = Contextual Permission Dependency Graph; CSSDG = Contextual Source and Sink Dependency Graph; ICFG = Interprocedural Control Flow Graph; AST = Abstract Syntax Tree; CPG = Code Property Graph; CFG = Control Flow Graph; PDG = Program Dependence Graph; CAG = Code Aggregate Graph; CDFG = Crucial Data Flow Graph; DFG = Data Flow Graph; SCPG = Simplified Code Property Graph; SPG = Slice Property Graph; VFG = Value Flow Graph; SDG = System Dependence Graph

REFERENCES

[1] J. C.S Santos, K. Tarrit, and M. Mirakhorli. A catalog of security architecture weaknesses. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017.
[2] I. Alexander. Misuse cases: use cases with hostile intent. *IEEE Software*, 2003.
[3] J. McDermott and C. Fox. Using abuse case models for security requirements analysis. In *Proceedings 15th Annual Computer Security Applications Conference (ACSAC'99)*, 1999.
[4] A. Shostack. *Threat modeling designing for security.* Wiley, 2014.
[5] S. T. Halkidis, N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides. Architectural risk analysis of software systems based on security patterns. *IEEE Transactions on Dependable and Secure Computing*, 2008.
[6] J. Ryoo, P. Laplante, and R. Kazman. A methodology for mining security tactics from security patterns. In *2010 43rd Hawaii International Conference on System Sciences*, 2010.
[7] L. Braz, E. Fregnan, G. Çalikli, and A. Bacchelli. Why don't developers detect improper input validation?'; drop table papers;–. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021.
[8] B. Arkin, S. Stender, and G. McGraw. Software penetration testing. *IEEE Security & Privacy*, 2005.
[9] B. Chess and G. McGraw. Static analysis for security. *IEEE Security & Privacy*, 2004.
[10] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *2010 23rd IEEE Computer Security Foundations Symposium*, 2010.
[11] G. Lin, W. Xiao, J. Zhang, and Y. Xiang. Deep learning-based vulnerable function detection: A benchmark. In *Information and Communications Security*, 2020.
[12] G. Renjith and S. Aji. Vulnerability analysis and detection using graph neural networks for android operating system. In *Information Systems Security*, 2021.
[13] N. Guo, X. Li, H. Yin, and Y. Gao. Vulhunter: An automated vulnerability detection system based on deep learning and bytecode. In *Information and Communications Security*, 2020.
[14] J. Chen, B. Liu, S. Cai, W. Wang, and S. Wang. Aidetectorx: A vulnerability detector based on tcn and self-attention mechanism. In *Dependable Software Engineering. Theories, Tools, and Applications*, 2021.

[15] B. Mosolygó, N. Vándor, P. Hegedűs, and R. Ferenc. A line-level explainable vulnerability detection approach for java. In *Computational Science and Its Applications – ICCSA 2022 Workshops*, 2022.
[16] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. De Vel, and P. Montague. Cross-project transfer representation learning for vulnerable function discovery. *IEEE Transactions on Industrial Informatics*, 2018.
[17] L. Cen, C. S. Gates, L. Si, and N. Li. A probabilistic discriminative model for android malware detection with decompiled source code. *IEEE Transactions on Dependable and Secure Computing*, 2015.
[18] A. Narayanan, L. Yang, L. Chen, and L. Jinliang. Adaptive and scalable android malware detection through online learning. In *2016 International Joint Conference on Neural Networks (IJCNN)*, 2016.
[19] A. Narayanan, M. Chandramohan, L. Chen, and Y. Liu. Context-aware, adaptive and scalable android malware detection through online learning (extended version), 2017.
[20] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models, 2017.
[21] P. Zegzhda, D. Zegzhda, E. Pavlenko, and G. Ignatev. Applying deep learning techniques for android malware detection. In *Proceedings of the 11th International Conference on Security of Information and Networks*, SIN '18, 2018.
[22] J. Allen, M. Landen, S. Chaba, Y. Ji, S. P. H. Chung, and W. Lee. Improving accuracy of android malware detection with lightweight contextual awareness. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, 2018.
[23] J. D. Koli. Randroid: Android malware detection using random machine learning classifiers. In *2018 Technologies for Smart-City Energy Security and Power (ICSESP)*, 2018.
[24] Z. Wang, J. Cai, S. Cheng, and W. Li. Droiddeeplearner: Identifying android malware using deep learning. In *2016 IEEE 37th Sarnoff Symposium*, 2016.
[25] N. Xie, F. Zeng, X. Qin, Y. Zhang, M. Zhou, and C. Lv. Repassdroid: Automatic detection of android malware based on essential permissions and semantic features of sensitive apis. In *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, 2018.
[26] A. Narayanan, M. Chandramohan, L. Chen, and Y. Liu. A multi-view context-aware approach to android malware detection and malicious code localization. *Empirical Software Engineering*, 2017.
[27] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *Computer Security - ESORICS 2014*, 2014.
[28] L. Braz, C. Aeberhard, G. Çalikli, and A. Bacchelli. Less is more: Supporting developers in vulnerability detection during code review. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, 2022.
[29] D. Vagavolu, K. C. Swarna, and S. Chimalakonda. A Mocktail of Source Code Representations. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.
[30] J. Chi, Y. Qu, T. Liu, Q. Zheng, and H. Yin. Seqtrans: Automatic vulnerability fix via sequence to sequence learning. *IEEE Transactions on Software Engineering*, 2023.
[31] F. Al Debeyan, T. Hall, and D. Bowes. Improving the performance of code vulnerability prediction using abstract syntax tree information. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2022, 2022.
[32] A. Sejfia and M. Schäfer. Practical automated detection of malicious npm packages. In *Proceedings of the 44th International Conference on Software Engineering*, May 2022.
[33] G. E. de P. Rodrigues, A. M. Braga, and R. Dahab. Using graph embeddings and machine learning to detect cryptography misuse in source code. In *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2020.
[34] G. Partenza, T. Amburgey, L. Deng, J. Dehlinger, and S. Chakraborty. Automatic identification of vulnerable code: Investigations with an ast-based neural network. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2021.
[35] A. Mester and Z. Bodó. Malware classification based on graph convolutional neural networks and static call graph features. In *Advances and Trends in Artificial Intelligence. Theory and Practices in Artificial Intelligence*, 2022.
[36] J. A. Harer et al. Automated software vulnerability detection with machine learning, 2018.
[37] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li. Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection. *Information and Software Technology*, 2021.
[38] I. Kalouptsoglou, M. Siavvas, D. Kehagias, A. Chatzigeorgiou, and A. Ampatzoglou. An empirical evaluation of the usefulness of word embedding techniques in deep learning-based vulnerability prediction. In *Security in Computer and Information Sciences*, 2022.
[39] R. Feng, Z. Yan, S. Peng, and Y. Zhang. Automated detection of password leakage from public github repositories. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022.
[40] N. Saccente, J. Dehlinger, L. Deng, S. Chakraborty, and Y. Xiong. Project achilles: A prototype tool for static method-level vulnerability detection of java source code using a recurrent neural network. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, 2019.
[41] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. VulDeePecker: A deep learning-based system for vulnerability detection. In *Proceedings 2018 Network and Distributed System Security Symposium*, 2018.

[42] V.-A. Nguyen, D. Q. Nguyen, V. Nguyen, T. Le, Q. H. Tran, and D. Phung. Regvd: Revisiting graph neural networks for vulnerability detection. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ICSE '22, 2022.

[43] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.*

[44] X. Cheng, G. Zhang, H. Wang, and Y. Sui. Path-sensitive code embedding via contrastive learning for software vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, 2022.

[45] S. M. Ghaffarian and H. R. Shahriari. Neural software vulnerability analysis using rich intermediate graph representations of programs. *Information Sciences*, 2021.

[46] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He. Smart contract vulnerability detection using graph neural network. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, 7 2020. Main track.

[47] P. Qian, Z. Liu, Q. He, R. Zimmermann, and X. Wang. Towards automated reentrancy detection for smart contracts based on sequential models. *IEEE Access*, 2020.

[48] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin. Vuldeelocator: A deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[49] B. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering. 01 2007.

[50] R.A. Kemmerer. Cybersecurity. In *25th International Conference on Software Engineering, 2003. Proceedings.*, 2003.

[51] X. Xia, Y. Wang, and Y. Yang. Source code vulnerability detection based on sar-gin. In *2021 2nd International Conference on Electronics, Communications and Information Technology (CECIT)*, 2021.

[52] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, 2014.

[53] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 2014.

[54] L. Buratti et al. Exploring software naturalness through neural language models, 2020.

[55] A. Bernstein and A. Kuleshov. Low-dimensional data representation in data analysis. In *Artificial Neural Networks in Pattern Recognition*, 2014.

[56] M. Grohe. Word2vec, node2vec, graph2vec, x2vec: Towards a theory of vector embeddings of structured data. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS'20, 2020.

[57] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space, 2013.

[58] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal. graph2vec: Learning distributed representations of graphs, 2017.

[59] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 2019.

[60] W. Ma, M. Zhao, E. Soremekun, Q. Hu, J. M. Zhang, M. Papadakis, M. Cordy, X. Xie, and Y. L. Traon. GraphCode2Vec: generic code embedding via lexical and program dependence analyses. In *Proceedings of the 19th International Conference on Mining Software Repositories*, Pittsburgh Pennsylvania, May 2022.

[61] S. Khan and S. Parkinson. *Review into State of the Art of Vulnerability Assessment using Artificial Intelligence.* 2018.

[62] V. H. S. Durelli, R. S. Durelli, S. S. Borges, A. T. Endo, M. M. Eler, D. R. C. Dias, and M. P. Guimarães. Machine learning applied to software testing: A systematic mapping study. *IEEE Transactions on Reliability*, 2019.

[63] M. T. Bin Nazim, M. J. H. Faruk, H. Shahriar, M. A. Khan, M. Masum, N. Sakib, and F. Wu. Systematic analysis of deep learning model for vulnerable code detection. In *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2022.

[64] H. P. Samoaa, F. Bayram, P. Salza, and P. Leitner. A systematic mapping study of source code representation for deep learning in software engineering. *IET Software*, 2022.

[65] Y. Yang, X. Xia, D. Lo, and J. Grundy. A survey on deep learning for software engineering, 2020.

[66] F. Ferreira, L. L. Silva, and M. T. Valente. Software engineering meets deep learning: A mapping study, 2020.

[67] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang. Software vulnerability detection using deep neural networks: A survey. *Proceedings of the IEEE*, 2020.

[68] T. Sonnekalb, T. S Heinze, and P. Mäder. Deep security analysis of program code. a systematic literature review.

[69] A. O. A. Semasaba, W. Zheng, X. Wu, and S. A. Agyemang. Literature survey of deep learning-based vulnerability analysis on source code. *IET Software*, 2020.

[70] S. M. Ghaffarian and H. R. Shahriari. Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey. *ACM Computing Surveys*, jul 2018.

[71] J. Wu. Literature review on vulnerability detection using nlp technology, 2021.

[72] Z. Chen and M. Monperrus. A literature study of embeddings on source code, 2019.

[73] Z. Kotti, R. Galanopoulou, and D. Spinellis. Machine learning for software engineering: A tertiary study. *ACM Comput. Surv.*, mar 2023.

[74] L. E. Lwakatare, A. Raj, J. Bosch, H. H. Olsson, and I. Crnkovic. A taxonomy of software engineering challenges for machine learning systems: An empirical investigation. In *Agile Processes in Software Engineering and Extreme Programming*, 2019.

[75] H. Hanif, M. H. Nizam Md Nasir, M. Faizal Ab Razak, A. Firdaus, and N. B. Anuar. The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches. *Journal of Network and Computer Applications*, 2021.

[76] M. Usman, M. A. Jan, X. He, and J. Chen. A survey on representation learning efforts in cybersecurity domain. *ACM Comput. Surv.*, 2019.

[77] M. Macas, C. Wu, and W. Fuertes. A survey on deep learning for cybersecurity: Progress, challenges, and opportunities. *Computer Networks*, 2022.

[78] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems*, 2019.

[79] H. H. Nguyen, N.-M. Nguyen, C. Xie, Z. Ahmadi, D. Kudendo, T.-N. Doan, and L. Jiang. Mando: Multi-level heterogeneous graph embeddings for fine-grained detection of smart contract vulnerabilities, 2022.

[80] Y. Zhuang, S. Suneja, V. Thost, G. Domeniconi, A. Morari, and J. Laredo. Software vulnerability detection via deep learning over disaggregated code graph representation, 2021.

[81] X. Cheng, H. Wang, J. Hua, M. Zhang, G. Xu, L. Yi, and Y. Sui. Static detection of control-flow-related vulnerabilities using graph embedding. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2019.

[82] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier. Toward large-scale vulnerability discovery using machine learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, 2016.

[83] J. Kronjee, A. Hommersom, and H. Vranken. Discovering software vulnerabilities using data-flow analysis and machine learning. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ARES '18, 2018.

[84] K. Cheng, G. Du, T. Wu, L. Chen, and G. Shi. Automated vulnerable codes mutation through deep learning for variability detection. In *2022 International Joint Conference on Neural Networks (IJCNN)*, 2022.

[85] T. Wu, L. Chen, G. Du, C. Zhu, N. Cui, and G. Shi. Inductive vulnerability detection via gated graph neural network. In *2022 IEEE 25th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, 2022.

[86] R. Rabheru, H. Hanif, and S. Maffeis. A hybrid graph neural network approach for detecting php vulnerabilities. In *2022 IEEE Conference on Dependable and Secure Computing (DSC)*, 2022.

[87] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H Jin. Vulcnn: An image-inspired scalable vulnerability detection system. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, 2022.

[88] Y. Li, S. Wang, and T. N. Nguyen. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, 2021.

[89] W. An, L. Chen, J. Wang, G. Du, G. Shi, and D. Meng. Avdhram: Automated vulnerability detection based on hierarchical representation and attention mechanism. In *2020 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, 2020.

[90] T. Wu, L. Chen, G. Du, C. Zhu, and G. Shi. Self-attention based automated vulnerability detection with effective data representation. In *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, 2021.

[91] J. Zeng, X. Nie, L. Chen, J. Li, G. Du, and G. Shi. An efficient vulnerability extrapolation using similarity of graph kernel of pdgs. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2020.

[92] A. Watson, E. Ufuktepe, and K. Palaniappan. Detecting software code vulnerabilities using 2d convolutional neural networks with program slicing feature maps. In *2022 IEEE Applied Imagery Pattern Recognition Workshop (AIPR)*, 2022.

[93] D. Zou, Y. Hu, W. Li, Y. Wu, H. Zhao, and H. Jin. mvulpreter: A multi-granularity vulnerability detection system with interpretations. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[94] L. K. Shar, H. Beng Kuan Tan, and L. C. Briand. Mining sql injection and cross site scripting vulnerabilities using hybrid program analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, 2013.

[95] G. Yan, S. Chen, Y. Bail, and X. Li. Can deep learning models learn the vulnerable patterns for vulnerability detection? In *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2022.

[96] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. 01 2014.

[97] L. Zhou, M. Huang, Y. Li, Y. Nie, J. Li, and Y. Liu. Grapheye: A novel solution for detecting vulnerable functions based on graph attention network. In *2021 IEEE Sixth International Conference on Data Science in Cyberspace (DSC)*, 2021.

[98] S. Suneja, Y. Zheng, Y. Zhuang, J. Laredo, and A. Morari. Learning to map source code to software vulnerability using code-as-a-graph, 2020.

[99] Y. Ding, S. Suneja, Y. Zheng, J. Laredo, A. Morari, G. Kaiser, and B. Ray. Velvet: a novel ensemble learning approach to automatically locate vulnerable statements. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022.

[100] X. Duan, J. Wu, S. Ji, Z. Rui, T. Luo, M. Yang, and Y. Wu. Vulsniper: Focus your attention to shoot fine-grained vulnerabilities. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, 7 2019.

[101] H. Wu, Z. Zhang, S. Wang, Y. Lei, B. Lin, Y. Qin, H. Zhang, and X. Mao. Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering*

(ISSRE), 2021.

[102] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security*, 2021.

[103] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, and X. Wang. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Transactions on Knowledge and Data Engineering*, 2023.

[104] Z. Zhang et al. Reentrancy vulnerability detection and localization: A deep learning based two-phase approach. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, 2023.

[105] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Trans. Softw. Eng. Methodol.*, apr 2021.

[106] J. Gear, Y. Xu, E. Foo, P. Gauravaram, Z. Jadidi, and L. Simpson. Scevd: Semantic-enhanced code embedding for vulnerability discovery. In *2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2022.

[107] Y. Wu, J. Lu, Y. Zhang, and S. Jin. Vulnerability detection in c/c++ source code with graph representation learning. In *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, 2021.

[108] Z. Liu, P. Qian, X. Wang, L. Zhu, Q. He, and S. Ji. Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion, 2021.

[109] L. K. Shar, L. C. Briand, and H. B. K. Tan. Web application vulnerability prediction using hybrid program analysis and machine learning. *IEEE Transactions on Dependable and Secure Computing*, 2015.

[110] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin. μvuldeepecker: A deep learning-based system for multiclass vulnerability detection. *IEEE Transactions on Dependable and Secure Computing*, 2021.

[111] W. Zheng, Y. Jiang, and X. Su. Vu1spg: Vulnerability detection based on slice property graph representation learning. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, 2021.

[112] Y. Xue, J. Guo, L. Zhang, and H. Song. Message passing graph neural networks for software security vulnerability detection. In *2022 International Conference on Computer Network, Electronic and Automation (ICCNEA)*, 2022.

[113] N. Sarbakysh and Z. Wang. A1bert: A language-agnostic graph neural network model for vulnerability detection. In *2023 8th International Conference on Data Science in Cyberspace (DSC)*, 2023.

[114] H. V. Nguyen, J. Zheng, A. Inomata, and T. Uehara. Code aggregate graph: Effective representation for graph neural networks to detect vulnerable code. *IEEE Access*, 2022.

[115] Z. Chen, S. Kommrusch, and M. Monperrus. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering*, 2023.

[116] Z. Yu, C. Theisen, L. Williams, and T. Menzies. Improving vulnerability inspection efficiency using active learning. *IEEE Transactions on Software Engineering*, 2021.

[117] A. Bagheri and P. Hegedűs. A comparison of different source code representation methods for vulnerability prediction in python. In *Quality of Information and Communications Technology*, 2021.

[118] D. Hin, A. Kan, H. Chen, and M. A. Babar. Linevd: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, 2022.

[119] M. Fu and C. Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, 2022.

[120] B. Mosolygó, N. Vándor, G. Antal, P. Hegedűs, and R. Ferenc. Towards a prototype based explainable javascript vulnerability prediction model. In *2021 International Conference on Code Quality (ICCQ)*, 2021.

[121] A. Mazuera-Rozo, A. Mojica-Hanke, M. Linares-Vásquez, and G. Bavota. Shallow or deep? an empirical study on detecting vulnerabilities using deep learning. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, 2021.

[122] H. Hanif and S. Maffeis. Vulberta: Simplified source code pre-training for vulnerability detection. In *2022 International Joint Conference on Neural Networks (IJCNN)*, 2022.

[123] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering*, 2021.

[124] T. H. M. Le, D. Hin, R. Croft, and M. A. Babar. Deepcva: Automated commit-level vulnerability assessment with deep multi-task learning. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ASE '21, 2022.

[125] T. H. M. Le, B. Sabir, and M. A. Babar. Automated software vulnerability assessment with concept drift. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019.

[126] X. Li, L. Wang, Y. Xin, Y. Yang, and Y. Chen. Automated vulnerability detection in source code using minimum intermediate representation learning. *Applied Sciences*, 2020.

[127] V. Nguyen, T. Le, T. Le, K. Nguyen, O. DeVel, P. Montague, L. Qu, and D. Phung. Deep domain adaptation for vulnerable code function identification. In *2019 International Joint Conference on Neural Networks (IJCNN)*, 2019.

[128] M.-J. Choi, S. Jeong, H. Oh, and J. Choo. End-to-end prediction of buffer overruns from raw source code via neural memory networks, 2017.

[129] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. E. Hassan. Finding a needle in a haystack: Automated mining of silent vulnerability fixes. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[130] G. Nguyen-Truong, H. J. Kang, D. Lo, A. Sharma, A. E. Santosa, A. Sharma, and M. Y. Ang. Hermes: Using commit-issue linking to detect vulnerability-fixing commits. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022.

[131] Y. Zhou, J. K. Siow, C. Wang, S. Liu, and Y. Liu. Spi: Automated identification of security patches via commits. *ACM Trans. Softw. Eng. Methodol.*, sep 2021.

[132] K. Filus, M. Siavvas, J. Domańska, and E. Gelenbe. The random neural network as a bonding model for software vulnerability prediction. In *Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2021.

[133] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, 2015.

[134] A. Xu, T. Dai, H. Chen, Z. Ming, and W. Li. Vulnerability detection for source code using contextual lstm. In *2018 5th International Conference on Systems and Informatics (ICSAI)*, 2018.

[135] N. Ziems and S. Wu. Security vulnerability detection using deep learning natural language processing. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2021.

[136] D. Cao, J. Huang, X. Zhang, and X. Liu. Ftclnet: Convolutional lstm with fourier transform for vulnerability detection. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2020.

[137] X. Wang, S. Wang, P. Feng, K. Sun, S. Jajodia, S. Benchaaboun, and f. Geck. Patchrnn: A deep learning-based system for security patch identification. In *MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM)*, 2021.

[138] C. Mamede, E. Pinconschi, R. Abreu, and J. Campos. Exploring transformers for multi-label classification of java vulnerabilities. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, 2022.

[139] F. R. Abdulhamza and R. J. S. Al-Janabi. Sql injection detection using 2d-convolutional neural networks (2d-cnn). In *2022 International Conference on Data Science and Intelligent Computing (ICDSIC)*, 2022.

[140] S. Ndichu, S. Ozawa, T. Misu, and K. Okada. A machine learning approach to malicious javascript detection using fixed length vector representation. In *2018 International Joint Conference on Neural Networks (IJCNN)*, 2018.

[141] M. Mimura and Y. Suga. Filtering malicious javascript code with doc2vec on an imbalanced dataset. In *2019 14th Asia Joint Conference on Information Security (AsiaJCIS)*, 2019.

[142] X. Zhou et al. Security code recommendations for smart contract. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2023.

[143] Z. Liu et al. Software vulnerability detection with gpt and in-context learning. In *2023 8th International Conference on Data Science in Cyberspace (DSC)*, 2023.

[144] Y. aand others Wei. Vulrep: Vulnerability repair based on inducing commits and fixing commits. *EURASIP Journal on Wireless Communications and Networking*, 2023.

[145] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. Codecmr: Cross-modal retrieval for function-level binary source code matching. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, 2020.

[146] R. Ferenc, P. Hegedűs, P. Gyimesi, G. Antal, D. Bán, and T. Gyimóthy. Challenging machine learning algorithms in predicting vulnerable javascript functions. In *2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, 2019.

[147] A. D. Sawadogo, T. F. Bissyandé, N. Moha, K. Allix, J. Klein, L. Li, and Y. L. Traon. Learning to catch security patches, 2020.

[148] I. Chowdhury and M. Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 2011. Special Issue on Security and Dependability Assurance of Software Architectures.

[149] L. Yang, X. Li, and Y. Yu. Vuldigger: A just-in-time and cost-aware tool for digging vulnerability-contributing changes. In *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, 2017.

[150] L. Kumar, C. Hota, A. Mahindru, and L. B. M. Neti. Android malware prediction using extreme learning machine with different kernel functions. In *Proceedings of the 15th Asian Internet Engineering Conference*, AINTEC '19, 2019.

[151] Z. Wang, J. Guo, and H. Li. Vulnerability feature extraction model for source code based on deep learning. In *2021 International Conference on Computer Network, Electronic and Automation (ICCNEA)*, 2021.

[152] M. Zagane, M. K. Abdi, and M. Alenezi. Deep learning for software vulnerabilities detection using code metrics. *IEEE Access*, 2020.

[153] S. Ganesh, T. Ohlsson, and F. Palma. Predicting security vulnerabilities using source code metrics. In *2021 Swedish Workshop on Data Science (SweDS)*, 2021.

[154] T.-Y. Chong, V. Anu, and K. Z. Sultana. Using software metrics for predicting vulnerable code-components: A study on java and python open source projects. In *2019 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, 2019.

[155] N. Medeiros, N. Ivaki, P. Costa, and M. Vieira. Vulnerable code detection using software metrics and machine learning. *IEEE Access*, 2020.

[156] M. Hasan, Z. Balbahaith, and M. Tarique. Detection of sql injection attacks: A machine learning approach. In *2019 International Conference on Electrical and Computing Technologies and Applications (ICECTA)*, 2019.

[157] M.-H. Tsai et al. Powerdp: De-obfuscating and profiling malicious powershell commands with multi-label classifiers. *IEEE Access*, 2023.

[158] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[159] A. Mahyari. A hierarchical deep neural network for detecting lines of codes with vulnerabilities. In *2022 IEEE 22nd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*, 2022.

[160] C. Thapa, S. I. Jang, M. E. Ahmed, S. Camtepe, J. Pieprzyk, and S. Nepal. Transformer-based language models for software vulnerability detection. In *Proceedings of the 38th Annual Computer Security Applications Conference*, ACSAC '22, 2022.

[161] Z. Tang, Q. Hu, Y. Hu, W. Kuang, and J. Chen. Sevuldet: A semantics-enhanced learnable vulnerability detector. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2022.

[162] S. Kim, J. Choi, M. E. Ahmed, S. Nepal, and H. Kim. Vuldebert: A vulnerability detection system using bert. In *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2022.

[163] S. Liu, G. Lin, L. Qu, J. Zhang, O. De Vel, P. Montague, and Y. Xiang. Cd-vuld: Cross-domain vulnerability discovery based on deep domain adaptation. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[164] J.-W. Liao, T.-T. Tsai, C.-K. He, and C.-W. Tien. Soliaudit: Smart contract vulnerability assessment based on machine learning and fuzz testing. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, 2019.

[165] S. Liu, G. Lin, Q.-L. Han, S. Wen, J. Zhang, and Y. Xiang. Deepbalance: Deep-learning and fuzzy oversampling for vulnerability detection. *IEEE Transactions on Fuzzy Systems*, 2020.

[166] N. Ashizawa, N. Yanai, J. P. Cruz, and S. Okamura. Eth2vec: Learning contract-wide code representations for vulnerability detection on ethereum smart contracts. In *Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure*, BSCI '21, 2021.

[167] R. Yan, X. Xiao, G. Hu, S. Peng, and Y. Jiang. New deep learning method to detect code injection attacks on hybrid applications. *Journal of Systems and Software*, 2018.

[168] D. Coimbra, S. Reis, R. Abreu, C. Păsăreanu, and H. Erdogmus. On using distributed representations of source code for the detection of c security vulnerabilities, 2021.

[169] G. Lin, J. Zhang, W. Luo, L. Pan, O. De Vel, P. Montague, and Y. Xiang. Software vulnerability discovery via learning multi-domain knowledge bases. *IEEE Transactions on Dependable and Secure Computing*, 2021.

[170] Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Çomak, and L. Karaçay. Vulnerability prediction from source code using machine learning. *IEEE Access*, 2020.

[171] H. Feng, X. Fu, H. Sun, H. Wang, and Y. Zhang. Efficient vulnerability detection based on abstract syntax tree and deep learning. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2020.

[172] M. Gu, H. Feng, H. Sun, P. Liu, Q. Yue, J. Hu, C. Cao, and Y. Zhang. Hierarchical attention network for interpretable and fine-grained vulnerability detection. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2022.

[173] Y. Mao, Y. Li, J. Sun, and Y. Chen. Explainable software vulnerability detection based on attention-based bidirectional recurrent neural networks. In *2020 IEEE International Conference on Big Data (Big Data)*, 2020.

[174] Y. He, H. Sun, and H. Feng. Ua-miner: Deep learning systems for expose unprotected api vulnerability in source code. In *2020 12th International Conference on Advanced Computational Intelligence (ICACI)*, 2020.

[175] P. Momeni, Y. Wang, and R. Samavi. Machine learning model for smart contracts security analysis. In *2019 17th International Conference on Privacy, Security and Trust (PST)*, 2019.

[176] X. Yan, S. Wang, and K. Gai. A semantic analysis-based method for smart contract vulnerability. In *2022 IEEE 8th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*, 2022.

[177] H. Shi, R. Wang, Y. Fu, Y. Jiang, J. Dong, K. Tang, and J. Sun. Vulnerable code clone detection for operating system through correlation-induced learning. *IEEE Transactions on Industrial Informatics*, 2019.

[178] W. Zheng, A. O. Abdallah Semasaba, X. Wu, S. A. Agyemang, T. Liu, and Y. Ge. Representation vs. model: What matters most for source code vulnerability detection. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021.

[179] M. Ceccato, C. D. Nguyen, D. Appelt, and L. C. Briand. Sofia: An automated security oracle for black-box testing of sql-injection vulnerabilities. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, 2016.

[180] X. Liu, X. Du, Q. Lei, and K. Liu. Multifamily classification of android malware with a fuzzy strategy to resist polymorphic familial variants. *IEEE Access*, 2020.

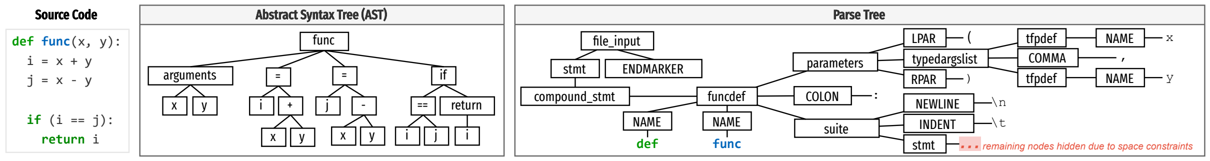
[181] E. Spirin, E. Bogomolov, V. Kovalenko, and T. Bryksin. Psiminer: A tool for mining rich abstract syntax trees from code. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021.

[182] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning distributed representations of code, 2018.
[183] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using parse tree validation to prevent sql injection attacks. In *Proceedings of the 5th International Workshop on Software Engineering and Middleware*, SEM '05, 2005.
[184] G. Lin, J. Zhang, W. Luo, L. Pan, and Y. Xiang. Poster: Vulnerability discovery with function representation learning from unlabeled projects. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, 2017.
[185] D. G. Fritz and R. G. Sargent. An overview of hierarchical control flow graph models. In *Proceedings of the 27th Conference on Winter Simulation*, WSC '95, 1995.
[186] S. Sinha. *Static and dynamic analysis of progams that contain arbitrary interprocedural control flow.* PhD thesis, 2002. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2023-03-03.
[187] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 1987.
[188] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, nov 2001.
[189] M. Weiser. Program slicing. *IEEE Transactions on software engineering*, (4), 1984.
[190] Y. Sui, X. Cheng, G. Zhang, and H. Wang. Flow2vec: Value-flow-based precise code embedding. *Proc. ACM Program. Lang.*, 2020.
[191] B. Steffen, J. Knoop, and O. Rüthing. The value flow graph: A program representation for optimal program transformations, 1990.
[192] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.
[193] Y. Zhang, X. Yu, Z. Cui, S. Wu, Z. Wen, and L. Wang. Every document owns its structure: Inductive text classification via graph neural networks, 2020.
[194] T. Xu and P. Zhou. Feature extraction for payload classification: A byte pair encoding algorithm. In *2022 IEEE 8th International Conference on Computer and Communications (ICCC)*, 2022.
[195] T. Kudo and J. Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing, 2018.
[196] X. Song, A. Salcianu, Y. Song, D. Dopson, and D. Zhou. Fast wordpiece tokenization, 2021.
[197] Q. V. Le and T. Mikolov. Distributed representations of sentences and documents, 2014.
[198] Meiliana, S. Karim, H. L. H. S. Warnars, F. L. Gaol, E. Abdurachman, and B. Soewito. Software metrics for fault prediction using machine learning approaches: A literature review with promise repository dataset. In *2017 IEEE International Conference on Cybernetics and Computational Intelligence (CyberneticsCom)*, 2017.
[199] I. Santos, F. Brezo, X. Ugarte-Pedrero, and P. G. Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, 2013. Data Mining for Information Security.
[200] B. Min, H. Ross, E. Sulem, A. P. B. Veyseh, T. H. Nguyen, O. Sainz, E. Agirre, I. Heinz, and D. Roth. Recent advances in natural language processing via large pre-trained language models: A survey, 2021.
[201] P. Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley, 2009.
[202] Y. Gui, Y. Wan, H. Zhang, H. Huang, Y. Sui, G. Xu, Z. Shao, and H. Jin. Cross-language binary-source code matching with intermediate representations, 2022.
[203] Z. Chen, S. Kommrusch, and M. Monperrus. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering*, January 2023.
[204] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection, 2023.
[205] R. L. Russell, L. Kim, L. H. Hamilton, T. Lazovich, J. A. Harer, O. Ozdemir, P. M. Ellingwood, and M. W. McConley. Automated vulnerability detection in source code using deep representation learning, 2018.
[206] T. Boland and P. E. Black. Juliet 1.1 c/c++ and java test suite. *Computer*, 2012.
[207] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, 2013.
[208] J. K. Siow, S. Liu, X. Xie, G. Meng, and Y. Liu. Learning program semantics with code representations: An empirical study. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022.
[209] Y. Hu, A. Huber, J. Anumula, and S. Liu. Overcoming the vanishing gradient problem in plain recurrent networks, 2019.
[210] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empir. Softw. Eng.*, 2009.
[211] J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 1960.

A EXAMPLES OF COMMONLY USED SOURCE CODE REPRESENTATIONS

A.1 Examples of Tree-Based Source Code Representations

Fig. 6 demonstrates how ASTs and parse trees differ for the same Python code. Notice how the parse tree retained every symbol in the code (*e.g.*, new lines and indentation) while the AST abstracted those away. Since parse trees are more verbose than ASTs, the work [179] that used it had a preprocessing step that removed from the tree nodes that were irrelevant to detect attacks (*e.g.*, specific user ids).



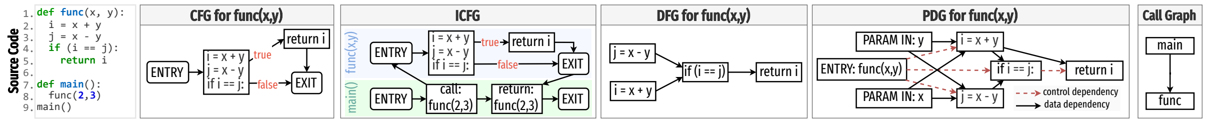
```
def func(x, y):
  i = x + y
  j = x - y

  if (i == j):
    return i
```

Fig. 6. Examples of tree representations for the same Python source code

A.2 Examples of Graph-Based Representations

Fig. 7 shows the CFG for the function `func(x,y)`. It has an *entry* and an *exit* basic block, to denote the start and end of the function execution, respectively. The *entry* block is connected to a basic block that has three instructions (*i.e.*, `i=x+y`, `j=x-y`, and `if i==j`). This basic block has two outgoing edges: one represents the flow when the if condition evaluates to *true*, and the other denotes the flow when it evaluates to *false*. After executing one of the branches, the execution flows into the *exit* block.



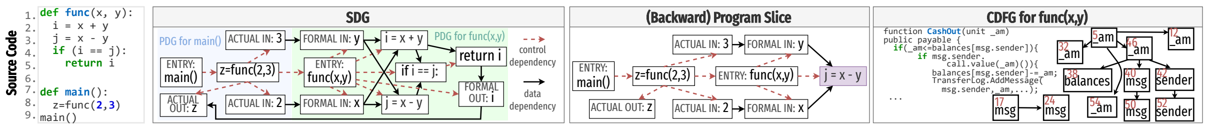
```
1. def func(x, y):
2.   i = x + y
3.   j = x - y
4.   if (i == j):
5.     return i
6.
7. def main():
8.   func(2,3)
9. main()
```

Fig. 7. Examples of graph-based representations (CFG, ICFG, DFG, PDG, and call graph) on the same Python code.

Additionally, Fig. 7 shows the ICFG for the entire program, including `func(x,y)` and `main()`. It has two *entry* blocks, one for `main`, where the program starts, and another for `func`. It also has two *exit* blocks which indicate the end of function execution for both `main` and `func`. The *entry* block for `main` is connected to the basic block that calls `func`, which, in turn, is connected to the callee's *entry* block. To capture the flow from the invoked function (*i.e.*, `func(x,y)`) back to its caller (*i.e.*, `main()`), this representation includes an edge from the callee's *exit* block to the caller's *return call site* block. In the same figure, it also shows the DFG for a Python function. There are two outgoing edges `i=x+y` to two statements that use the variable `i`. Similarly, there is an edge from `j=x-y` to `if i==j` because the expression is using the variable `j`.

Fig. 7 has an example of a PDG for a Python function (`func(x,y)`), where the dashed and straight lines represent control dependencies and data dependencies, respectively. The PDG starts with an *entry* node for this procedure. It has two *param in* nodes to represent the function's parameters (`x` and `y`). For the statements in lines 2-4 to execute, the function's entry must have executed. As such, there is a control dependency edge from the *entry* to the nodes representing these lines of code: `i=x+y`, `j=x-y`, and `if i == j`. Since the statements in lines 2 and 3 define variables that are used in the `if` condition, there is a data dependency edge from these variable assignments statements to the `if` condition. There is also a data dependency edge from the `i=x+y` to the `return` statement, since this `return` node uses the variable `i`. Given that the `return` statement only executes when the if condition evaluates to true, there is a control dependency between the **`if`** node and the **`return`** node. Finally, Fig. 7 shows the call graph for the Python code snippet provided in it.

In Fig. 8, the SDG for the entire program is shown. The SDG starts with the *entry* node for `main()`, which is the entrypoint function in the call graph. The first entry node from `main` has a control flow edge to `func`. Two *actual in* nodes have data flow edges, as the data is initialized from `main` and flows to `func`. There are also two *actual in* nodes, which define the parameters entering the procedure `func(x,y)`. The *entry* node in `func` has a control flow edge to the statement nodes which define the variables `i` and `j`, and the two *param in* nodes have a data flow edge to those same two nodes; `i=x+y` and `j=x-y`. These statement nodes then both have a data flow to the predicate node `if i == j`. The *entry* node has a control dependency edge to this node and the *formal in* nodes. Finally, this predicate node has a control flow edge to the `return i` block, along with `i=x+y` node, except this node has a data dependency edge. The dashed line in this figure depict control dependencies and the solid line depicts data dependencies. Fig. 8 also shows an example of a *backward program slice* over the SDG with the slice criterion $\langle j, line\ 3\rangle$. The nodes that have control or data dependencies to the node of interest (*i.e.*, `j = x - y`) are part of the resulting program slice. In Figure 8, we also see our crucial nodes for a Crucial Data Flow Graph: any nodes that have to deal with `msg`, `sender`, `balances`, and `_am`. These nodes receive or compute data, and can be responsible for a reentrancy vulnerability.



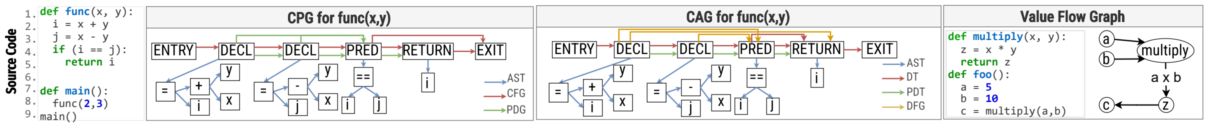
```
1. def func(x, y):
2.   i = x + y
3.   j = x - y
4.   if (i == j):
5.     return i
6.
7. def main():
8.   z=func(2,3)
9. main()
```

```
function CashOut(unit _am)
public payable {
  if(_am<=balances[msg.sender]){
      if msg.sender.
            call.value(_am)()){
          balances[msg.sender]-=_am;
          TransferLog.AddMessage(
            msg.sender,_am,...);
 ...
```

Fig. 8. Examples of graph representations (SDG, Backward Program Slices & CDFG)

Figure 9 demonstrates the CPG for `func()`. We have the *entry* node at the start, demonstrating the entrance into the function. From the entry node, we have a *control flow* edge to the *declaration* node, which then branches to show the *AST* nodes and edges for the line `i=x+y`. We then have another control flow edge from the declaration node to another declaration node. This node branches down to show the AST nodes and edges for the line `j=x-y`. From this node, we have a control flow edge to a *predicate node*. We also have *PDG* edges from the first and second declaration node to this predicate node, as the predicate node depends on the data from the two declaration nodes. The predicate node also has AST nodes and edges to show the line `i==j`. From here, we have a control flow edge to a *return* statement. We also have a PDG edge to this node from the predicate node, as the return statement depends on the result from the predicate edge. We finally have a control flow edge to the *exit* node from the return. The exit node also has a control flow edge from the predicate node because if the predicate evaluates to false, the program terminates. The same figure also demonstrates an example of a CAG for the function `func()`. We start with the *entry* node, which has a *dominator tree* edge to the `DECL` node. The `DECL` node then has *AST* nodes and edges representing the line `i=x+y`. From the `DECL` node, we have a *dominator tree* and *post dominator tree* edge to another `DECL` node. Once again, the `DECL` node has *AST* nodes and edges representing the line `j=x-y`. Next, we have a *dominator tree* and *post dominator tree* edge to the `PRED` node from the `DECL` node. Moreover, we have a *data dependence* edge from the `DECL` node defining `i` and the `DECL` node defining `j` nodes, as the if statement needs the data from both of these nodes to execute. The `PRED` node has a *dominator tree* and *post dominator tree* edge to the `RETURN` node. The `RETURN` node has a *data dependence* edge from the `DECL` node that defines i, since the return statement depends on that node's data. Finally, we have a dominator tree edge to the `EXIT` node.

Fig. 9 shows the VFG for the code snippet provided. We start with the nodes a and b. These two nodes have an edge into the `multiply(x,y)` node, as the values from a and b are passed into this function. We then have an edge to node `z`, which represents the multiplication operation on a and b. Finally, `z` has an edge connecting it to `c`, which stores the value from the multiplication operation.



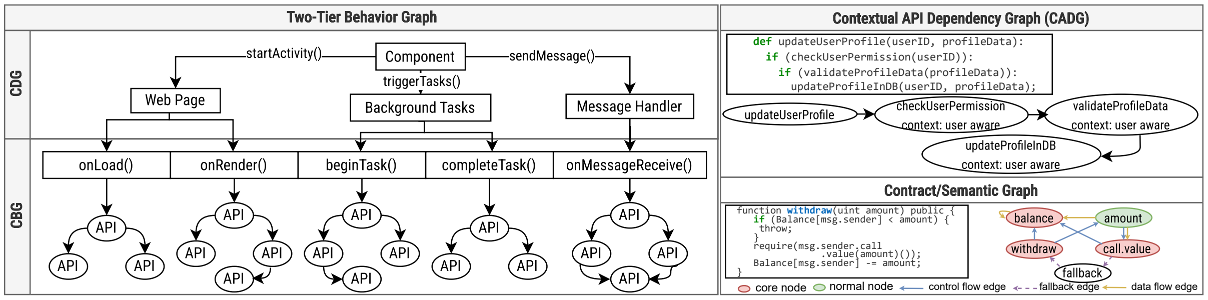
```
def func(x, y):
  i = x + y
  j = x - y
  if (i == j):
    return i

def main():
  func(2,3)
main()
```

```
def multiply(x, y):
  z = x * y
  return z
def foo():
  a = 5
  b = 10
  c = multiply(a,b)
```

Fig. 9. Examples of graph representations (CPG, CAG, and VFG) on the same code snippet

Fig. 10 shows an example of the CDG which starts at the *component* node. We then have three edges: one for `startActivity()`, which leads to a *web page* node, one for `triggerTasks()` which leads to the *background tasks* node, and finally a `sendMessage()` edge which leads to a *message handler* node. Below the CDG, there is an example of the CBG, as a continuation of the larger Web Application Behavior Graph. The *web page* node from the CDG leads to two different nodes in the CBG: `onLoad()` and `onRender()`. Both of these nodes lead to *API* nodes, which can represent any of the aforementioned node types. This pattern continues: the *background tasks* node from the CDG leads to the `beginTask()` node and `complexTask()` node, which then lead to *API* nodes representing the further activities of the APIs in the program. Finally, the *message handler* node from the CDG leads to `onMessageReceive()` node in the CBG, which leads to other API nodes that represent their functionality within the program.



```
def updateUserProfile(userID, profileData):
  if (checkUserPermission(userID)):
    if (validateProfileData(profileData)):
      updateProfileInDB(userID, profileData);
```

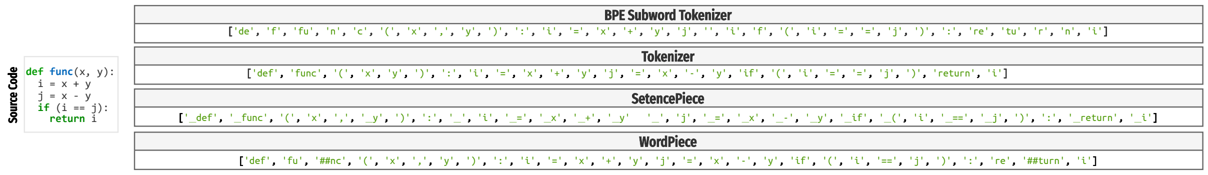
```
function withdraw(uint amount) public {
   if (Balance[msg.sender] < amount) {
    throw;
   }
   require(msg.sender.call
            .value(amount)());
   Balance[msg.sender] -= amount;
}
```

Fig. 10. Examples of CDG, CBG and CADG.

Figure 10 also gives an example of the CADG for the code snippet shown. We start with the function `updateUserProfile`, and have a control flow edge to the security related node `checkUserPermission` with the context *user aware*. From here, we flow to another security related node, `validateProfileData` with the same context: *user aware*. Finally, we flow to `updateProfileDB` with the same *user aware* context. The same figure also shows a Contract/Semantic graph for the provided code snippet. The `withdraw` and `balance` are core nodes, and the `call.value` variable is a core node because they could be the cause for a vulnerability. The `amount` variable is a normal node, because it does not directly contribute to a security issue. The `withdraw` invocation has control flow edges to the `balance` invocation, and the `amount` variable. The `balance` node as a data flow edge to itself since it is accessing data from itself, and receives or depends on data from the variable `amount`. `call.value` also receives data from `amount` and thus there is a data flow edge from `amount` to `call.value`. There are control flow edges from `withdraw` to `balance` and `amount` because it invokes these two nodes. After `amount` is called, then `call.value` is invoked, thus resulting in a control flow edge from `amount` to `call.value`. Finally, there is fallback edge from `call.value` to the `fallback` node, and from the `fallback` node back to `withdraw`.

A.3 Lexical Representations

Figure 11 provides an example of the resulting tokens from different tokenization techniques, including a standard tokenizer, BPE subword tokenization, SentencePiece, and WordPiece.



```
def func(x, y):
  i = x + y
  j = x - y
  if (i == j):
    return i
```

```
['de', 'f', 'fu', 'n', 'c', '(', 'x', ',', 'y', ')', ':', 'i', '=', 'x', '+', 'y', 'j', '', 'i', 'f', '(', 'i', '=', '=', 'j', ')', ':', 're', 'tu', 'r', 'n', 'i']
```

```
['def', 'func', '(', 'x', 'y', ')', ':', 'i', '=', 'x', '+', 'y', 'j', '=', 'x', '-', 'y', 'if', '(', 'i', '=', '=', 'j', ')', 'return', 'i']
```

```
['_def', '_func', '(', 'x', ',', '_y', ')', ':', '_', 'i', '_=', '_x', '_+', '_y'   '_', 'j', '_=', '_x', '_-', '_y', '_if', '_(', 'i', '_==', '_j', ')', ':', '_return', '_i']
```

```
['def', 'fu', '##nc', '(', 'x', ',', 'y', ')', ':', 'i', '=', 'x', '+', 'y', 'j', '=', 'x', '-', 'y', 'if', '(', 'i', '==', 'j', ')', ':', 're', '##turn', 'i']
```

Fig. 11. Examples of different tokenizing algorithms