Graph-Based Profiling of Dependency Vulnerability Remediation

Fernando Vera Buschmann[1], Palina Pauliuchenka[1], Ethan Oh[1], Bai Chien Kao[2], Louis DiValentin[2], and David A. Bader[1]

[1] Department of Data Science,
New Jersey Institute of Technology, Newark, NJ, USA
{fv54,pp272,eo238,bader}@njit.edu
[2] Accenture PLC {bai.chien.kao, louis.divalentin}@accenture.com

Abstract. This research presents an enhanced Graph Attention Convolutional Neural Network (GAT) tailored for the analysis of open-source package vulnerability remediation. By meticulously examining control flow graphs and implementing node centrality metrics—specifically, degree, norm, and closeness centrality—our methodology identifies and evaluates changes resulting from vulnerability fixes in nodes, thereby predicting the ramifications of dependency upgrades on application workflows. Empirical testing on diverse datasets reveals that our model challenges established paradigms in software security, showcasing its efficacy in delivering comprehensive insights into code vulnerabilities and contributing to advancements in cybersecurity practices. This study delineates a strategic framework for the development of sustainable monitoring systems and the effective remediation of vulnerabilities in open-source software.

Keywords: Graph Attention Convolutional Neural Network (GAT), Package Vulnerability Analysis, open source, package upgrade, Knowledge Graph, Node Centrality Metrics, Cybersecurity, Deep Learning Applications, Network Analysis, Code Vulnerability Mitigation.

1 Introduction

The increasing complexity of software systems and the reliance on third-party libraries have significantly heightened the importance of vulnerability analysis in open-source packages. Despite progress in identifying and mitigating vulnerabilities, significant gaps remain in understanding the interdependencies and impact of remediation efforts. This study focuses on three real-world applications: a Python-based data processing package, a Java-based web application, and a mixed-language analytics tool. By analyzing these diverse codebases, we aim to demonstrate the versatility and effectiveness of our graph-based approach.

Preventing and analyzing vulnerabilities is crucial for averting cyber attacks and protecting programs and components [31]. The growing complexity and abstraction of dependencies and version management increase the likelihood of vulnerabilities within application code in this dynamic, continuously updating

world [15]. During code development, it is essential to identify and address the weakest link or the most apparent vulnerability and to pursue continuous solutions with each code update [14]. To effectively protect against cyber attacks, three steps are often followed: 1) know the vulnerabilities, 2) understand their impact on the application, and 3) act on them in depth.

Vulnerability remediation in open source packages is vital for the open-source community and user security. This process involves identifying vulnerabilities, reporting them, and analyzing their impact [2]. Open source maintainers then triage these vulnerabilities and bugs, prioritize updates, and release new package versions that fix the affected functions and remove bugs.

Various tools facilitate the discovery of vulnerabilities in dependencies, including CVEfixes, which replicates the comprehensive database from the U.S. National Vulnerability Database (NVD). Static and dynamic code analysis capabilities, such as CodeQL used by the GitHub community, can identify and modify functions to discover new vulnerabilities [25, 26]. Software Composition Analysis (SCA) techniques create a Software Bill of Materials (SBOM) listing the specific versions of dependencies used. This SBOM is correlated with existing vulnerability repositories to identify vulnerabilities in each dependency, targeting them for upgrades to newer versions where issues are fixed. However, upgrading dependencies can sometimes cause compilation errors or functional issues in the application [12, 29].

Understanding the impact of dependency version changes, along with functional interconnections and network structures, is crucial for estimating the likelihood of specific package upgrades causing breaking changes in application code. Given the distinct and adaptable nature of applications, tools that generalize across different applications are needed to provide insights into upgrade complexity. Thus, we turn to knowledge graphs [13, 24]. Constructing a graph allows us to comprehend interactions between functions within the code. In this research, we represent functions as entities in a graph space, constructing an Inter Procedural Control Flow graph of function interactions. This enables examination of the application code and dependencies, with the caller-callee relationship as the connecting link [32]. Observing and analyzing these interactions assists in identifying the impact of dependency changes on application workflow and downstream effects. This relational understanding supports the automation of package upgrades with minimal functionality impact, directly translating to improved security by enabling automated processes to upgrade vulnerable package versions with minimal disruptions.

Understanding the impact of changed function nodes between two application versions using a graph necessitates a profound comprehension of the network's topology and connectivity attributes [33]. This knowledge is indispensable for identifying the significance of affected nodes to the network's overall operation and making informed decisions on upgrade impacts [4]. This process involves analyzing centrality, density, connected components, degree assortativity coefficient, and pathways. Such comprehensive evaluations facilitate strategic decision-making for optimally mitigating vulnerabilities, ensuring a robust and

secure network architecture. This is particularly critical in software, where functions exhibit complex nested dependencies and feedback loops.

The remainder of this paper is organized as follows: Section 2 outlines our proposed approach. Section 3 reviews the background and related work. Section 4 elaborates on the methodology, including the problem statement and workflow. Section 5 describes the experimental setup and presents the results and analysis. Finally, Section 6 discusses the findings and provides the conclusion.

2 Proposed Approach

The central issue addressed in this paper is the challenge of accurately identifying and mitigating vulnerabilities in open-source packages without causing functional disruptions. Our methodology leverages a Graph Attention Network (GAT) [28] to analyze the interdependencies of software components and predict the impact of remediation efforts on software workflows. This approach is formalized by examining control flow graphs and applying node centrality metrics [5].

2.1 Problem Statement

The primary problem is to develop a model that can pinpoint changed nodes and forecast the impact of dependency updates. This involves analyzing the structural and functional attributes of the code to ensure that upgrades do not introduce new vulnerabilities or disrupt existing functionalities [6, 26].

2.2 Workflow

Our workflow involves four main stages: data collection, graph construction, vulnerability analysis, and impact assessment. Each stage is designed to incrementally build a comprehensive understanding of the software's vulnerability landscape.

2.3 Domain-Specific Example

To illustrate our methodology, consider an open-source Python-based data processing package. In this context: - **Nodes**: Represent functions and classes within the package. For example, a node might represent a function responsible for data encryption. - **Features**: Include degree centrality, which indicates the number of direct connections a function has, and closeness centrality, which measures how quickly a function can interact with other functions in the network.

When a vulnerability is identified in a critical function (e.g., the encryption function), our approach maps the control flow graph for the application and analyzes each function for differences between the current and remediated version of the open source package. The GAT model is then applied to analyze how the changed functions impact related functions and the overall system performance.

By doing so, we can predict potential disruptions and ensure that remediation efforts do not affect existing functionalities.

This concrete scenario bridges the gap between the high-level overview of our methodology and its technical implementation, enhancing the clarity and applicability of our approach.

3 Background and Related Work

Traditional methods of package function analysis primarily rely on static and dynamic analysis techniques. While static analysis provides a broad overview, it often lacks the context-specific insights gained from dynamic runtime analysis [19]. Additionally, these techniques typically focus on the implementations of functions within the application, rather than vulnerabilities in the application's dependencies. Some scanning techniques and vendors address these dependency vulnerabilities by upgrading to a version where the vulnerability is resolved and recompiling the code. If the remediated version causes compilation failures or unit test failures, further code updates are required to ensure compatibility. Understanding the effects of upgrading to a remediated package version remains a complex and opaque process.

Inter Procedural Control flow graph analysis is a useful tool for understanding software behavior, allowing developers to easily visualize the flow of execution within a program. By reducing abstract and complex code into manageable graphs, it provides information about the structure of the program and its possible vulnerabilities and affected functions. This analysis is crucial to optimize code performance and ensure strong cybersecurity measures. In our case, we enhance the inter procedural control flow graph with a knowledge graph related to the use of algorithms and measures focused on large-scale analysis.

Graph Attention Networks (GAT) have emerged as a promising tool in similar analytical contexts. Originally conceptualized by Veličković et al. [28], GATs leverage the attention mechanism to provide node-specific contextual insights, enhancing the accuracy of feature representation in graph-structured data. This has been effectively utilized in various domains, including bioinformatics, social network analysis, and natural language processing [30]. In package function analysis, GATs offer an innovative opportunity to address the limitations of traditional methods. They provide a scalable approach to analyze the intricate interrelationships and dependencies among package functions represented as nodes in a graph. This method aligns with recent trends in utilizing deep learning techniques for software engineering challenges, as documented in several contemporary studies [8].

4 Methodology

This section is outlining key definitions and delving into the critical elements that underpin our methodological approach. A portion of our research is the deployment of a modified Graph Attention Network (GAT), an advanced model that is being specifically designed to analyze feature code interactions and study the importance of changed functions represented as nodes in the graph.

4.1 Modified Graph Attention Neural Network

The adaptation of the GAT model is motivated by the lack of current visibility into the effects of upgrading package versions, which directly affects developers ability to stay on top of package vulnerabilities originating from dependencies. Unlike traditional GAT models based solely on norm [28], our modified version incorporates essential node centrality metrics (degree centrality, norm, and closeness) to evaluate the importance and interrelationships of various functions within the code, with a particular focus on identifying and evaluating vulnerable nodes. This enhancement increases the ability of GATs to provide a more refined analysis of the relational dynamics within the code, offering a comprehensive perspective on the impact of changes in dependency versions and the potential risk of breakage resulting from changed code.

Rationale for Modification

Utilizing the Knowledge Graph for network analysis, particularly emphasizing on graph attention Neural Network (GAT) as defined in the work of Veličković et al. [28], is enabling us to train extensive datasets for the purpose of evaluating and quantifying the significance of nodes within a network. This base model is facilitating accurate detection of nodes based on both transductive methods (such as Cora, Citeseer, Pubmed) and inductive methodologies. Presented as a convolutional-like neural network operating on knowledge graph-structured data, this model assigns an importance metric to nodes within their neighborhoods.

The input to our layer consists of a set of node features, $h = \{\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_N\}$, with $\mathbf{h}_i \in \mathbb{R}^F$, where N denotes the number of nodes and F the number of features per node. This layer is producing a new set of node features, $h' = \{\mathbf{h}'_1, \mathbf{h}'_2, \ldots, \mathbf{h}'_N\}$, with a potentially different cardinality F'. A shared linear transformation, parameterized by a weight matrix $\mathbf{W} \in \mathbb{R}^{F' \times F}$, is applied to each node. The GAT then performs a self-attention mechanism a, yielding attention coefficients e_{ij} as: $e_{ij} = a(\mathbf{W}\mathbf{h}_i, \mathbf{W}\mathbf{h}_j)$

These coefficients indicate the importance of node j's features to node i. The GAT model is assessing the first-order neighbors of i, necessitating a normalization of coefficients across all node features j using the softmax function:

$\alpha_{ij} = \text{softmax}_j(e_{ij}) = \exp(e_{ij}) / \sum_{k \in \mathcal{N}_i} \exp(e_{ik})$

it is calculated the coefficient most relevant of each feture of node using the norm. It identified as α_{ij}. Replacing equation e_{ij} in equation α_{ij}

$$\alpha_{ij} = \frac{\exp\left(\text{LeakyReLU}\left(a^\top [\mathbf{W}h_i | \mathbf{W}hj]\right)\right)}{\sum k \in \mathcal{N}(i) \exp\left(\text{LeakyReLU}\left(a^\top [\mathbf{W}h_i | \mathbf{W}h_k]\right)\right)} \tag{1}$$

In line with the original GAT model [28], with the attention mechanism a as a single-layer feedforward neural network, we define a parameterized weight matrix $\mathbf{a} \in \mathbb{R}^{2F'}$ and apply the LeakyReLU nonlinearity (negative input slope of 0.2). $a^\top$ is a learnable weight vector in the single-layer feedforward neural network which constitutes the attention mechanism. $\mathcal{N}(i)$ denotes the neighborhood of node i in the graph. $\|\|$ represents the concatenation operation.$e_{ij} = a(\mathbf{W}\mathbf{h}_i, \mathbf{W}\mathbf{h}_j)$

This process calculates the most relevant coefficient for each feature of a node using the norm, identified as α_{ij}. To enhance the focus and applicability of these coefficients, particularly in code analysis, we are introducing a modification β_{ij} to highlight the robustness and criticality of all nodes. This modification encompasses functions such as degree centrality, the norm, and closeness centrality metrics, relevant to nodes i and j. Consequently, the revised attention coefficient, α'_{ij}, is defined as: $\alpha'_{ij} = \alpha_{ij} \cdot \beta_{ij}$

This novel approach is proving instrumental in evaluating the importance of specific functions within their neighborhoods, offering insights into the criticality of an affected function in a code structure. The modified model enables the generation of a normalized score ranging from 0 to 1, which reflects the importance of nodes based on metrics such as degree, norm, and centrality. This scoring system facilitates a more nuanced understanding of node significance within the graph structure, particularly in the context of package code analysis.

4.2 Mapping and Mitigating Vulnerability in a Code Development

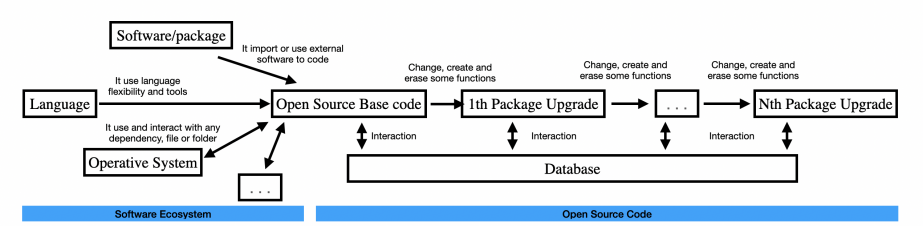

Fig. 1: In this figure, we are providing a conceptual description of the actors considered behind the open source. We are taking into account a generic software ecosystem of code from key elements such as the operating system, language, and software, packages or others that enable the code to function and connect with the real world. A code under development typically begins with the Open Source Base code (primary functions) that is interacting with the repository, the operating system, and other components of the ecosystem. This base code is undergoing updates Nth times to enhance its functionality. It is crucial to emphasize that within each component, there could exist certain flaws which might potentially manifest as vulnerabilities within the code.

Vulnerabilities are recorded in the NVD repositories from various sources during the development process. These can be inherent from the base of the code (fig.1) to the programming language used, integrated through specific packages or software, or due to operating system flaws [7, 9]. These vulnerabilities are remediated by standardizing community-driven coding practices [**?**], like SAST and DAST scanning, and bug reporting. Nevertheless, vulnerability mapping is continuously evolving, with new vulnerabilities emerging or being discovered over time, as well as new tools and methods for finding them [3,27]. Thus, it is proving

beneficial to maintain an updated, dynamic, and comprehensive connection map, which serves as a guide to pinpoint the source and address these vulnerabilities swiftly and precisely.

From the foundational layer of code, the language, environment, configurations [2], and built-in packages [1] emerge. The core code develops from this foundation, forming the program's heart with its principal functions. An initial version of the code is committed, and subsequent updates add functionality and features, resolving errors and introducing new functions and packages. Each update may integrate previous functions into the core or leave them as branches. Vulnerabilities can appear in any update or trace back to the base code, and a vulnerable fragment can extend its impact across updates to critical branches, potentially affecting the entire application's core structure. Understanding the network of functions within the code is crucial to mitigate risks and ensure updates do not compromise the application's integrity.

4.3 Knowledge Graph in Cybersecurity

According to Hogan [11], a debate is ongoing regarding the precise definition of a Knowledge Graph, yet consensus exists about its remarkably high adaptability. In the context of this discussion, the knowledge graph G is defined as $G = (E, R, S)$, where E represents the entity (node), R symbolizes the relationship (edge), and S denotes relationship facts (node-node relationship). A triplet constitutes a typical form of knowledge representation within this framework. Entities, serving as foundational elements of the Knowledge Graph, encompass a wide range of classifications, such as collections, categories, object types, and thing categories (e.g., domain, host, etc.). Relationships interconnect these entities to formulate the graph's structure, while attributes encapsulate features and parameters, exemplified by entities like google.com, windows, and similar.

To construct a dataset, it is necessary to study the relationships existing between functions through their callee or caller interactions. A database is generated considering the node entities E_i, their relations R_i, and their relation S_i, which can be a call or a caller. This approach enables the generation of a knowledge graph containing the identified entities. The data structure provides critical information, encompassing the function's path, its name, and whether it has been modified in the latest update. Additionally, it indicates whether the function is vulnerable and specifies its role as either a callee or a caller.

4.4 Building the Dataset

To build a comprehensive dataset, we compare subsequent versions of code where upgraded package versions are intended to remediate vulnerabilities in open source packages [1]. We use the open source software Syft to generate the software bill of materials (SBOM) for the current version of the target application source code. Curl fetches vulnerability data for individual package versions in the SBOM from the Open Source Vulnerabilities (OSV) database, which provides an accessible query interface for all known dependency vulnerabilities. This

information is mapped to the SBOM to identify existing vulnerabilities in the dependencies. If a package with a vulnerability is detected, we search for the updated version of the package and clone it. Using CodeQL, a semantic code analysis engine, we model different versions of the code and construct control flow graphs of the execution paths for the repository using both the vulnerable and fixed packages. Tree-sitter performs incremental analysis, constructs, and maintains a syntax tree, and builds the dataset. The impact of a single package upgrade is measured by comparing the control flow for matching functions in the inter procedural graph of the application before and after the upgrade. The updated version control flow graph serves as the base for the knowledge graph, with changed or affected nodes marked, along with nodes causing compilation errors. Multiple upgrades are performed in two sets for each repository: one set with graphs introducing errors resulting in code breakage, and another set with graphs not impacting code functionality.

In this research, a preliminary approach is being used, initially focusing on three application source repositories; The first case in a code base comprising 9621 features with 27 vulnerabilities in its broken update; the second case with 19,569 functions and 3 vulnerabilities. Subsequently, in Cases 3 (15908 functions and 6 vulnerabilities) and 4 (16095 functions and no vulnerabilities), an application is observed with a similar order of magnitude but different dependencies regarding the base functions.

5 Results and Analysis

Our experimental setup involves analyzing three diverse codebases in Python and Java, employing Graph Attention Networks (GAT) for vulnerability assessment [28]. The results, underscore the efficacy of our approach in accurately identifying critical vulnerabilities and predicting the impact of remediation efforts. For instance, in one case study, our model effectively pinpointed a vulnerable encryption function and forecasted its implications on the data processing workflow, thereby demonstrating its practical utility in real-world scenarios.

In this section, we discuss the methods of data analysis and describe the metrics we will use. Once the graph structure is built, we can utilize tools like NetworkX in Python for small-scale analysis [20] or Arachne for large-scale projects [21]. In this case, NetworkX is used exclusively to facilitate calculation and analysis, given the volume of data (less than 20,000 nodes). This analysis adopts a two-pronged approach. Firstly, it involves an exploratory examination of graph connectivity, focusing on the distribution and other metrics that provide insights into the differences between package upgrades that break dataflow and those that do not. This includes evaluating the structural design and robustness of the network, ensuring that each functional node and its connections contribute to the overall integrity and efficiency of the system. Secondly, we analyze the modified GAT scores to obtain a normalized measure of the interaction of vulnerabilities as nodes within the graph. The model provides an advanced mechanism to measure the importance of nodes within a network, allowing for

a targeted strategy for classifying and assessing vulnerabilities and simplifying the assessment of remediation effectiveness. The practical benefits of this approach include prioritizing development efforts, improving software integrity, formulating strategic planning initiatives, and deepening knowledge about software architecture. This comprehensive strategy not only improves the immediate security posture of software systems but also lays the foundation for long-term sustainable software development and maintenance practices.

The results highlight the model's ability to maintain system stability while addressing vulnerabilities, underscoring its potential for broader application in cybersecurity.

5.1 Graph analysis

It is crucial when analyzing the knowledge graph created with the data set to measure the network and understand the meaning and relevance of the entities and relationships within it [5]. It is necessary to quantify the connectivity, the paths and their strength through the centrality analysis of each objective element.

Preliminary Graph Analysis

We are constructing a knowledge graph, denoted as $G = (\hat{E}, \hat{R}, \hat{S})$, where $\hat{E} = V$ signifies the entities or nodes. Our primary focus is on the callee-caller relationship, considered as directional edges. This simplifies the triplet $(\hat{R}, \hat{S}) = E$ into the graph structure $G = (V, E)$ [5], aligning with the GAT model [28].

This knowledge graph features a dynamic representation of functions, with vertices n and directed edges m. Each edge $e \in E$ is assigned a specific weight $w(e)$, reflecting the significance and connectivity strength between functions. Paths within the graph are sequences of edges $\langle u_i, u_{i+1} \rangle$, where $u_0 = s$ and $u_l = t$ denote the start and end vertices. The distance between two vertices s and t, represented as $d(s, t)$, is indicated by the shortest paths σ_{st}. We quantify the number of shortest paths traversing a specific vertex v using $\sigma_{st}(v)$, consistent with Bader et al. [5].

Degree centrality: We are measuring the degree centrality of a vertex v, denoted as $\deg(v)$, to quantify the extent of interactions a node has within its neighborhood. This metric reflects the node's importance based on the number of caller-callee connections it maintains [5].

The Norm: By definition of the Euclidean norm of a vector used by velivckovic et al [28] in GATs model, $\mathbf{x} \in \mathbb{R}^n$ as: $\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$ where $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ represents a vector in an n-dimensional real space, and x_i corresponds to the i-th element of the vector.

Closeness Centrality: Closeness centrality measures the degree of proximity of a node to all other nodes in the graph, based on distance. For any node n, its closeness centrality is calculated as the average length of the shortest path from n to every other node. A node with higher closeness centrality is more centrally located in the network, indicating greater importance or influence within

the network's structure. This metric is determined by the inverse of the sum of the shortest distances from the node to all other nodes [5, 18]:

$$CC(v) = \frac{1}{\sum_{u \in V} d(v,u)}$$

In our implementation, this translates to assessing how interconnected a specific function is relative to the rest of the functions within the code.

Betweenness Centrality: Betweenness centrality quantifies how often a node appears on the shortest paths between other nodes, acting as a critical bridge within the network [5]. In software systems, a function with high betweenness centrality is pivotal in the flow of information or processes, significantly influencing other functions. The pairwise dependency $\delta_{st}(v)$, representing the fraction of shortest paths between nodes s and t passing through node v, is defined as: $\delta_{st}(v) = \sigma_{st}(v)/\sigma_{st}$ This leads to the formulation of betweenness centrality for a node v, where $s, v, t \in V$:

$$BC(v) = \sum_{s \neq v \neq t} \delta_{st}(v)$$

Betweenness centrality reflects a node's ability to control information or resource flow by bridging the shortest paths between nodes. Nodes with high betweenness centrality are essential for network connectivity, facilitating communication and interactions by being part of numerous shortest paths connecting various node pairs.

Connected components: Connected components are defined as subgraphs where any two vertices are connected by paths, without external connections [10]. Our research defines a connected component as a set of nodes where each node can access all other nodes within the same set. This concept is crucial for analyzing network structures, detecting isolated clusters, and examining connectivity. The study distinguishes between undirected graphs, where a connected component comprises the largest set of interconnected nodes, and directed graphs, which include strongly connected components defined by bidirectional paths between all pairs of nodes. Our approach leverages the concept of connected components to gain insights into the network's architecture. This analysis is pivotal in identifying potential vulnerabilities or areas of improvement within the network, particularly in cybersecurity and software engineering. By understanding the formation and interaction of these components, we can devise more effective strategies for network optimization and vulnerability mitigation.

Clustering Coefficient: The clustering coefficient for a node v, denoted as C_i, assesses the likelihood of connectivity between two randomly chosen neighbors of this node. This measure indicates the number of triangles in which the i-th node participates, normalized by the maximum possible number of such triangles. We compute the average clustering coefficient by averaging these individual values across all nodes in the graph:

$$C_i = \frac{2t_i}{k_i(k_i - 1)},$$

where t_i is the number of triangles around node i, and k_i is the degree of node i. As this coefficient approaches 1, it suggests increasing completeness of the graph with a predominant cohesive component. Higher coefficients indicate triadic closure, observed in denser graphs with prevalent triangular formations [22]. The average clustering for a graph is calculated as:

$$\bar{C}_i = \frac{1}{n} \sum_{i=1}^{n} C_i.$$

This metric provides an overall indication of the degree of clustering within the network, reflecting how closely nodes tend to cluster together, thus offering insights into the network's structural density and connectivity patterns.

Degree Assortativity Coefficient: We measure the degree assortativity coefficient to quantify the tendency of nodes in a network to connect with other nodes of similar degree, providing insights into assortative or disassortative mixing [17]. This metric determines whether high-degree nodes are more likely to connect with other high-degree nodes or with low-degree nodes. The coefficient ranges from -1 to 1, where values close to 1 indicate assortative mixing and values close to -1 indicate disassortative mixing. A value around 0 indicates no particular connectivity preference. The degree assortativity coefficient r is defined as:

$$r = \frac{\sum_{jk} jk(e_{jk} - q_j q_k)}{\sigma_q^2},$$

where e_{jk} is the fraction of edges connecting nodes of degree j and k, q_j is the distribution of the remaining degrees of nodes, and σ_q^2 is the variance of q. This coefficient is crucial for understanding the structural tendencies of the network and how nodes preferentially form connections based on their degrees.

Cyclomatic complexity Cyclomatic complexity serves as a quantitative measure for evaluating the number of linearly independent paths within a code, estimating the program's complexity [6]. This metric is crucial for understanding the intricacy and structural complexity of a program [23]. Adapting McCabe's definition [16], the cyclomatic complexity $V(G)$ of a control flow graph G is defined as: $V(G) = E - N + 2P$ where E represents the number of edges, N signifies the number of nodes, and P stands for the number of connected components. This measure provides insight into potential paths and decision points in a program's structure, helping to understand software maintainability and areas for refactoring. Cyclomatic complexity is thus a practical tool for guiding the development and maintenance of robust and efficient software systems.

5.2 Dataset Results and Interpretation

This preliminary study analyzes three different Python and Java applications, leveraging open source packages with control flow graphs of base versions, broken post-upgrade, and non-broken post-upgrade. CodeQL analysis flags critical features, identifying changes that cause build failures. The goal is to understand

how the structural and functional attributes of the functions in the graph affect the probability of code breaking, especially due to modifications and new features or patches to fix vulnerabilities.

CASE 1: (Table 1) We analyzed an application code base with 9626 functions, identifying 27 as critical, where remediation upgrades caused code breakage. Post-update, we observed an increase in the number of nodes and average degree, while network density remained constant. The number of connected components increased, and the degree assortativity coefficient remained negative and constant. Notably, the degree assortativity coefficient for the Non-Broken Upgrade was much lower than that for the Broken Upgrade, and closeness centrality was higher for Non-Broken Upgrades.

	Base	**Non Broken Upgrade**		**Broken Upgrade**	
Number of unique functions	9621	9621		9621	
nodes kind		**Unchanged**	**Changed**	**Unchanged**	**Changed**
Number of nodes	9621	9614	19	9613	23
Number of edges	15186	15178	19	15176	24
Average degree	3.1568	3.1575	2.0	3.1574	2.087
Density	0.0003281	0.0003285	0.1111	0.0003285	0.09486
Num. of connected components	327	327	2	327	2
Average clustering	0.06809	0.06815	0.0	0.06815	0.0
Degree assortativity coefficient	-0.08515	-0.08519	-0.3996	-0.08521	-0.3464
Avg. betweenness centrality	0.00036	0.000361	1.63973	0.00036	2.13699
Avg. closeness centrality	0.15857	0.15864	0.11057	0.15871	0.0909
Cyclomatic Complexity	6219	6218	4	6217	5

Table 1: (Case1) Comparative analysis of code metrics from code base, non-broken update and broken upgrade.

CASE 2: (Table 2) We analyzed an application code base with 19,569 functions and 37,615 caller-recipient interactions, identifying 3 functions as critical in their failed update, causing code breakage. Both upgrades significantly affected the number of functions, although the total number of functions remained the same. Cyclomatic complexity decreased after both upgrades, while the number of connected components increased. The degree assortativity coefficient for the Non-Broken Upgrade was lower than that for the Broken Upgrade, as was the density of the changed nodes, while closeness centrality was higher for Non-Broken Upgrades.

CASE 3: (Table 3) We analyzed an application code base with 15,908 functions and 29,449 caller-recipient interactions, identifying 6 critical functions in the upgrade. Post-upgrade, cyclomatic complexity decreased, and the number of connected components increased. The degree assortativity coefficient for the Non-Broken Upgrade was higher than that for the Broken Upgrade, while the density of changed nodes remained lower and closeness centrality was higher for Non-Broken Upgrades.

	Base	**Non Broken Upgrade**		**Broken Upgrade**	
Number of unique functions	19569	19569		19569	
nodes kind		**Unchanged**	**Changed**	**Unchanged**	**Changed**
Number of nodes	19569	17635	2629	17255	3338
Number of edges	37615	33180	4586	32234	5850
Average degree	3.8443	3.7630	3.4888	3.7362	3.5051
Density	0.00019646	0.00021339	0.00132754	0.00021654	0.00105037
Number of connected components	440	426	80	429	97
Average clustering	0.05552	0.05505	0.04415	0.05467	0.04482
Degree assortativity coefficient	-0.07772	-0.08047	-0.12601	-0.07853	-0.12377
Avg. betweenness centrality	0.00018	0.00019	0.000124	0.00019	0.00012
Avg. closeness centrality	0.17790	0.17686	0.18617	0.18716	0.17640
Cyclomatic Complexity	18926	16397	2117	15837	2706

Table 2: (CASE 2) Comparative analysis of code metrics from code base, non-broken upgrade and broken upgrade.

	Base	**Non Broken Upgrade**		**Broken Upgrade**	
Number of unique functions	15908	15908		15908	
nodes kind		**Unchanged**	**Changed**	**Unchanged**	**Changed**
Number of nodes	15908	15553	603	15127	1015
Number of edges	29449	28616	912	27695	1816
Average degree	3.7024	3.6798	3.0249	3.6617	3.5783
Density	0.00023275	0.00023661	0.00502471	0.00024208	0.00352892
Number of connected components	337	341	19	335	28
Average clustering	0.05004	0.04834	0.06402	0.04688	0.05890
Degree assortativity coefficient	-0.10958	-0.11123	-0.10286	-0.11203	-0.13705
Avg. betweenness centrality	0.00022	0.00022	0.00018	0.00022	0.00022
Avg. closeness centrality	0.18028	0.18032	0.17876	0.18049	0.17684
Cyclomatic Complexity	14215	13745	347	13238	857

Table 3: (CASE 3) Comparative analysis of code metrics from code base, non-broken update and broken upgrade

Statistical Differences between Base and Subgraphs: Given the consistent differences in closeness centrality across the three cases, we further explored these variations. Conducting a package upgrade generates two subgraphs of the control flow graph: functions affected by the upgrade and those not affected. We hypothesize that the subgraphs resulting from broken package upgrades are statistically different from randomly drawn subgraphs due to higher concentrations of vulnerabilities. This hypothesis is based on the premise that vulnerabilities are likely to disrupt the normal connectivity patterns, which can be captured through centrality metrics.

To test this hypothesis, we used the closeness centrality values for each node and applied T-tests and Kolmogorov-Smirnov (K-S) tests to evaluate the differences in means and distributions between the subgraphs. The T-test assesses whether the average closeness centrality of nodes in the affected subgraph differs significantly from the baseline, while the K-S test compares the overall distributions to identify broader variations in connectivity patterns. These statistical

methods are appropriate because they provide a robust framework for detecting changes in node centrality, which is crucial for understanding the impact of upgrades on the network structure.

For Broken Case 1, we observed a T-statistic of -4.881 (p-value = 0.0012) and a K-S statistic of 0.6916 (p-value = 8.16e-05), indicating significant deviations in closeness centrality between changed nodes and all nodes. In contrast, Broken Case 2 had a T-statistic of 8.399 (p-value = 6.47e-17) and a K-S statistic of 0.1161 (p-value = 1.38e-27), suggesting less variation but still significant differences. Broken Case 3 showed significant differences with a K-S statistic of 0.2176 (p-value = 9.81e-34) and a T-statistic of -2.349 (p-value = 0.019). These results validate the hypothesis by confirming that the mean values and distributions of closeness centrality are statistically different between the subgraphs resulting from upgrades that break functionality and the base repositories.

The results from the three cases analyzed here exhibit a trend where cyclomatic complexity tends to keep or decrease as upgrades are made to package. This observation suggests a relationship between the resolution of vulnerabilities and the simplification of code structure that may justify further exploration. This ongoing analysis is crucial for understanding the dynamic nature of package vulnerabilities and their impact on overall code complexity.

Modified GAT Results The next step of the analysis attempts to analyze the interconnectedness of the critical functions causing the package upgrades to fail. When we apply the modified GAT model Fig. 2, the scores obtained. It is necessary to see each case as a specific case. For this, an average GAT score was obtained for each case. Having a high GAT score above average indicates that the vulnerabilities are more critical and necessary since the network depends more on them.

	Case 1	**Case 2**	**Case 3**
NC	27	3	6
MSC	0.5295	0.3662	0.5045
mSC	0.2461	0.1254	0.1260
ASC	0.4359	0.2185	0.3228
AGS	0.4287	0.3785	0.3153

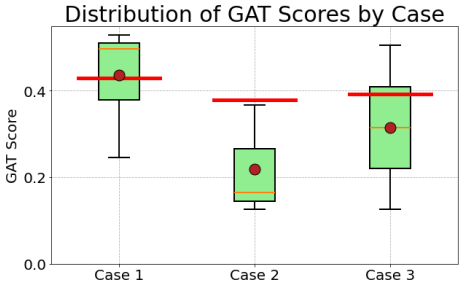

Fig. 2: NC: number of Critical Function, MSFC:Max Score Critical, mSFC:Min Score Critical, ASC:Avr Score Critical, and AG:Avr Gat Score for all nodes(Red line)

Principal Component Analysis (PCA) and t-Distributed Stochastic Neighbor Embedding (t-SNE) [28] are dimensionality reduction techniques applied alongside GATs to enhance visualization and interpretability as we do in the Case 2 Fig.5. While PCA projects data into a lower-dimensional space preserving vari-

ance, t-SNE focuses on maintaining local relationships, making it particularly useful for visualizing high-dimensional data generated by GATs in a way that highlights patterns and relationships within the graph structure.

The enhanced GAT Score significantly improves our ability to discern the connectivity and importance of critical functions within a system's context. By employing a normalized approach, where scores closer to 1 denote higher importance, we gain nuanced insights into specific vulnerabilities. This is exemplified in CASE 1, where the average GAT score for the entire graph is 0.4287, with most of the 27 critical functions scoring above this value (see Fig. 2). In CASE 2, the GAT scores for all critical functions are below the average, with a maximum critical function score of 0.3662, while the average score for the entire graph is 0.3785, indicating lower criticality. These scores integrate a weighted combination of degree, norm, and centrality metrics, rather than relying solely on connectivity. This holistic approach allows us to identify functions that, despite having lower connectivity, hold substantial significance within the network's overall architecture. Such insights underscore the complexity of network dynamics and the crucial role of advanced analytical tools in unveiling the intricate interplay of functions within a software system.

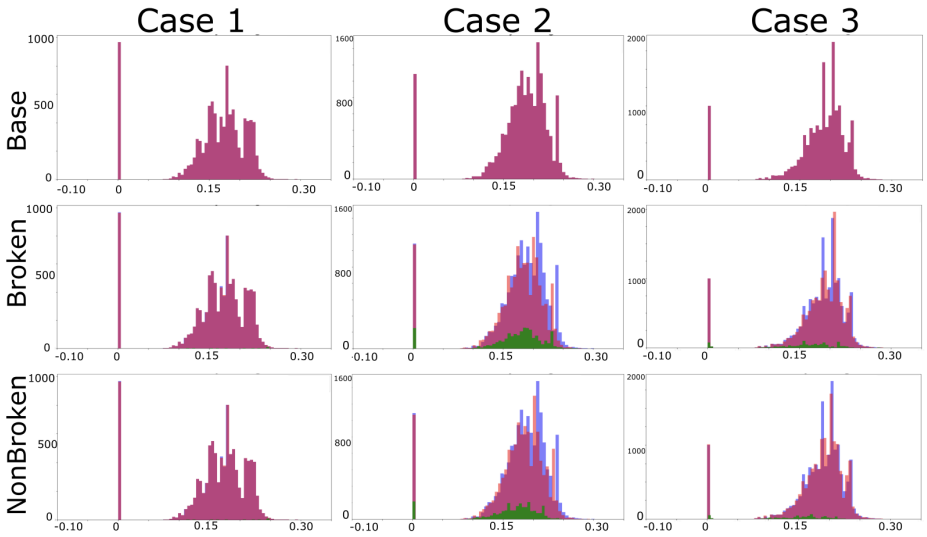

Fig. 3: Closeness Centrality Histogram, y-axes counts and x-axes Closeness Centrality. Cases 1, 2, 3, and 4 with their respective base code, broken upgrade, and non-broken upgrade as applicable. Light blue indicates the centrality of all nodes, light red represents unchanged nodes, and green denotes changed nodes.

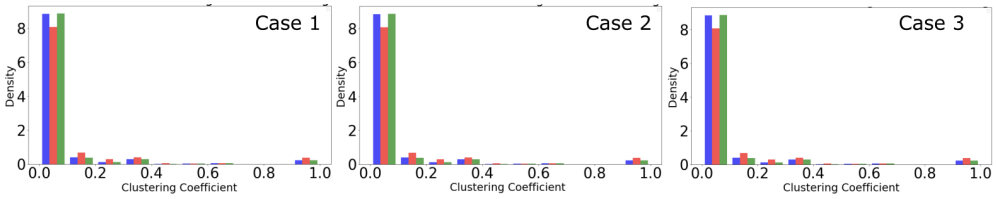

Fig. 4: Normalized cases of clustering coefficient histogram. Blue: All nodes, Red: Changed nodes, Green: Non changed nodes.

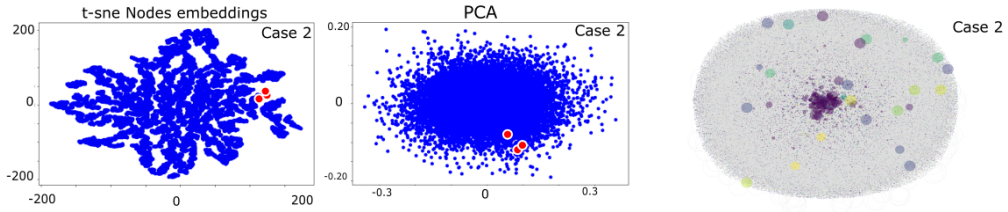

Fig. 5: Visualization of communities for Case 2, using t-SNE and PCA applied to a modified Graph Attention Network (GAT) as usually worked in GAT data analysis,The red dot is the representation of the vulnerability in their respective spaces. The third figure is a graphical representation of the communities within the graph, each distinguished by a unique color.

6 Discussion

The results of our extensive analysis of several code bases reveal an intricate dynamic between package vulnerabilities and the complexity of the code. This study contributes to the current discourse in package development by highlighting the nuanced relationship between the vulnerabilities after the publication and evolution of the code.This requires not only immediate code updates, but also a deeper understanding of the underlying dynamics of these vulnerabilities.

Our analysis of CASES 1, 2, and 3 reveals several key insights into the impact of vulnerability resolution on code structure. We observed a trend of decreased or consistent cyclomatic complexity when comparing base cases with broken and intact vulnerability resolutions, indicating that addressing vulnerabilities simplifies the code structure and aligns with best practices for maintainable and less error-prone packages. However, managing the complexity of package vulnerabilities remains challenging. Notably, the number of connected components increased from the base case to the non-broken update, suggesting that remediation efforts improve connectivity and robustness. Despite this, even minor vulnerabilities can affect the entire application, highlighting the need for a comprehensive approach to understanding and mitigating the impacts of package updates. This complexity is further underscored by cross-centrality graphs (3, 4), which show significant variations in node centrality, emphasizing the critical importance of detailed graph-based analyses for managing software vulnerabilities.

In addition, our analysis sheds light on the connectivity patterns of the function within the code bases. The coefficient of negative degree supply consistently observed in multiple studies indicates that the code functions tend to connect with a diverse variety of other functions, instead of predominantly linked to similar functionalities. This diversity in connectivity patterns has deep implications to understand how vulnerabilities could spread through a code base and affect their general integrity.

This ambiguity highlights a critical gap in current understanding and requires more research. It emphasizes the importance of a strategic approach to code updates, where essential functionalities such as configuration are refined and maintained, instead of being completely eliminated.

The significance of the modified GAT score is underscored by our observations of the wide dispersion of nodes and low density within the analyzed graphs. This dispersion requires a nuanced approach to understanding the role of each node, focusing on both its connections and the paths traversing through it. The modified GAT model provides a normalized view of how interconnected a function is concerning its degree, norm, and closeness centrality metrics. This comprehensive perspective is crucial for effective vulnerability management and software maintenance, highlighting critical nodes that warrant prioritized attention. By integrating GAT score insights with node propagation and density observations, stakeholders are equipped with a powerful tool to identify and address significant vulnerabilities, optimizing resource allocation and enhancing software security and reliability. This blend of quantitative measures and attention-based analysis represents a significant advancement in strategically mitigating vulnerabilities and strengthening software infrastructure.

Our study underscores the dynamic nature of package vulnerabilities and their impact on code complexity, emphasizing the need for continuous vigilance and strategic intervention throughout the package development lifecycle. By hypothesizing that broken package upgrade subgraphs differ statistically from randomly drawn subgraphs due to higher vulnerability concentrations, we validated our approach using closeness centrality values for each node and comparative T-tests and Kolmogorov-Smirnov tests, confirming the reliability of our vulnerability assessments. To illustrate the versatility and effectiveness of our graph-based methodology, we analyzed three real-world applications: a Python-based data processing package, a Java-based web application, and a mixed-language analytics tool. This comprehensive analysis demonstrates the applicability of our methods across different programming environments, highlighting critical functions within each application that contribute to increased vulnerability risks. By integrating insights from our modified Graph Attention Network (GAT) with node propagation and density observations, we offer valuable strategic planning for vulnerability mitigation, ultimately enhancing software security and reliability across diverse software ecosystems.

7 Conclusion

This paper embarks on a comprehensive exploration, leveraging the intricate capabilities of knowledge graphs to delve into the dynamics of opensource package function networks. Central to our investigation is the identification and analysis of vulnerable functions, where we scrutinize their interactions and assess the impact of their mitigation on the codebase. Our research reveals a notable insight: targeted remediation of specific vulnerabilities tends to preserve the overall network's connectivity, underscoring the package structure's inherent resilience.

During our analysis, we noted a pattern of decreasing vulnerabilities following successive package updates, prompting a pivotal inquiry: are these vulnerabilities conclusively resolved, or do they subtly embed themselves into subsequent features? This ambiguity signals a compelling need for further work into the lifecycle of vulnerabilities within package development, promising to enrich the discourse in this field significantly.

At the heart of our methodology is the odified Graph Attention Network (GAT), especially its attention mechanism. Through the integration of node-centric metrics—such as degree centrality, norm, and closeness centrality—our approach refines the network's ability to discern detailed aspects of the graph's architecture and the nuances of node characteristics. This methodological advancement facilitates a nuanced portrayal of the network, yielding a comprehensive understanding of node interrelations and their significance.

Furthermore, our investigation brings to light the existence of latent vulnerabilities within the most critical segments of the code, initially perceived as flawless. These covert vulnerabilities represent significant security risks, with the potential to compromise vital components, including databases and core functionalities. To address these issues, we advocate for an in-depth and ongoing code analysis from its inception. Utilizing knowledge graphs as both historical and dynamic monitoring tools enables proactive surveillance of vulnerabilities.

For future works, our research direction will focus on methodological improvements, particularly in dissecting the interconnectivity between functions. The number of case analyzed was small with significant complexity, necessitating additional application repositories for comprehensive analysis to draw more robust conclusions. By evaluating various aspects such as variable types, execution times, and functional dependencies, we aim to unravel the importance of specific functions within the network. This comprehensive strategy is designed to offer deeper insights into the structural integrity and vulnerabilities of software systems, thereby making a substantial contribution to enhancing package security and dependability. Our endeavors are geared towards the development of robust software systems capable of navigating the complexities of contemporary cyber threats, marking a significant stride forward in the realm of package vulnerability analysis and cybersecurity.

Acknowledgment

This research was supported in part by NSF grant CCF-2109988.

References

1. Manar Alanazi, Abdun Mahmood, and Mohammad Jabed Morshed Chowdhury. SCADA vulnerabilities and attacks: A review of the state-of-the-art and open issues. *Computers & Security*, page 103028, 2022.
2. Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. Empirical analysis of security vulnerabilities in Python packages. *Empirical Software Engineering*, 28(3):59, 2023.
3. Raghavendra Rao Althar, Debabrata Samanta, Manjit Kaur, Dilbag Singh, and Heung-No Lee. Automated risk management based software security vulnerabilities management. *IEEE Access*, 10:90597–90608, 2022.
4. Ashwin Arulselvan, Clayton W Commander, Lily Elefteriadou, and Panos M Pardalos. Detecting critical nodes in sparse graphs. *Computers & Operations Research*, 36(7):2193–2200, 2009.
5. David A. Bader and Kamesh Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *2006 International Conference on Parallel Processing (ICPP'06)*, pages 539–550, 2006.
6. Christof Ebert, James Cain, Giuliano Antoniol, Steve Counsell, and Phillip Laplante. Cyclomatic complexity. *IEEE Software*, 33(6):27–29, 2016.
7. Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. A C/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 508–512, New York, NY, USA, 2020. Association for Computing Machinery.
8. Görkem Giray. A software engineering perspective on engineering machine learning systems: State of the art and challenges. *Journal of Systems and Software*, 180:111031, 2021.
9. Katerina Goseva-Popstojanova and Andrei Perhinschi. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*, 68:18–33, 2015.
10. Lifeng He, Xiwei Ren, Qihang Gao, Xiao Zhao, Bin Yao, and Yuyan Chao. The connected-component labeling problem: A review of state-of-the-art algorithms. *Pattern Recognition*, 70:25–43, 2017.
11. Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia D'Amato, Gerard De Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. Knowledge Graphs. *ACM Computing Surveys*, 54(4):1–37, Jul 2021.
12. Nasif Imtiaz, Seaver Thorn, and Laurie Williams. A comparative study of vulnerability reporting by software composition analysis tools. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2021.
13. Yan Jia, Yulu Qi, Huaijun Shang, Rong Jiang, and Aiping Li. A practical approach to constructing a Knowledge Graph for cybersecurity. *Engineering*, 4(1):53–60, 2018.
14. Kai Liu, Fei Wang, Zhaoyun Ding, Sheng Liang, Zhengfei Yu, and Yun Zhou. Recent progress of using Knowledge Graph for cybersecurity. *Electronics*, 11(15):2287, 2022.
15. Kai Liu, Fei Wang, Zhaoyun Ding, Sheng Liang, Zhengfei Yu, and Yun Zhou. A review of Knowledge Graph application scenarios in cybersecurity. *arXiv preprint arXiv:2204.04769*, 2022.

16. Thomas J McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, (4):308–320, 1976.
17. Mark EJ Newman. Mixing patterns in networks. *Physical review E*, 67(2):026126, 2003.
18. UJ Nieminen. On the centrality in a directed graph. *Social Science Research*, 2(4):371–378, 1973.
19. Namyong Park, Andrey Kan, Xin Luna Dong, Tong Zhao, and Christos Faloutsos. Estimating node importance in Knowledge Graphs using Graph Neural Networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery Data Mining*, pages 596–606, 2019.
20. Edward L Platt. *Network Science with Python and NetworkX Quick Start Guide: Explore and Visualize Network Data Effectively*. Packt Publishing Ltd, 2019.
21. Oliver Alvarado Rodriguez, Zhihui Du, Joseph Patchett, Fuhuan Li, and David A Bader. Arachne: An Arkouda package for large-scale graph analytics. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2022.
22. Jari Saramäki, Mikko Kivelä, Jukka-Pekka Onnela, Kimmo Kaski, and Janos Kertesz. Generalizations of the clustering coefficient to weighted complex networks. *Physical Review E*, 75(2):027105, 2007.
23. Mir Muhammad Suleman Sarwar, Sara Shahzad, and Ibrar Ahmad. Cyclomatic complexity: The nesting problem. In *Eighth International Conference on Digital Information Management (ICDIM 2013)*, pages 274–279. IEEE, 2013.
24. Leslie F Sikos. Cybersecurity Knowledge Graphs. *Knowledge and Information Systems*, pages 1–21, 2023.
25. Sherri Sparks, Shawn Embleton, Ryan Cunningham, and Cliff Zou. Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 477–486. IEEE, 2007.
26. Tamás Szabó. Incrementalizing production CodeQL analyses. *arXiv preprint arXiv:2308.09660*, 2023.
27. Ángel Jesús Varela-Vaca, Diana Borrego, María Teresa Gómez-López, Rafael M Gasca, and A German Márquez. Feature models to boost the vulnerability management process. *Journal of Systems and Software*, 195:111541, 2023.
28. Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph Attention Networks. *arXiv preprint arXiv:1710.10903*, 2017.
29. Boming Xia, Tingting Bi, Zhenchang Xing, Qinghua Lu, and Liming Zhu. An empirical study on software bill of materials: Where we stand and the road ahead. *arXiv preprint arXiv:2301.05362*, 2023.
30. Feng Xia, Xin Chen, Shuo Yu, Mingliang Hou, Mujie Liu, and Linlin You. Coupled attention networks for multivariate time series anomaly detection. *IEEE Transactions on Emerging Topics in Computing*, 2023.
31. Zhihao Yan and Jingju Liu. A review on application of Knowledge Graph in cybersecurity. In *2020 International Signal Processing, Communications and Engineering Management Conference (ISPCEM)*, pages 240–243. IEEE, 2020.
32. Kailong Zhu, Yuliang Lu, Hui Huang, Lu Yu, and Jiazhen Zhao. Constructing more complete control flow graphs utilizing directed gray-box fuzzing. *Applied Sciences*, 11(3):1351, 2021.
33. Afra Zomorodian. Computational topology. *Algorithms and Theory of Computation Handbook*, 2(3), 2009.