VIVID: A Novel Approach to Remediation Prioritization in Static Application Security Testing (SAST)

Naeem Budhwani
Accenture
Montreal, Canada
naeem.budhwani@gmail.com

Mohammad Faghani
Accenture
Toronto, Canada
mrfaghani@gmail.com

Hayden Richard
Accenture
Nashville, USA
haydenrichard411@gmail.com

***Abstract*—Static Application Security Testing (SAST) enables organizations to detect vulnerabilities in code early; however, major SAST platforms do not include visual aids and present little insight on correlations between tainted data chains. We propose VIVID - Vulnerability Information Via Data flow - a novel method to extract and consume SAST insights, which is to graph the application's vulnerability data flows (VDFs) and carry out graph theory analysis on the resulting VDF directed graph. Nine metrics were assessed to evaluate their effectiveness in analyzing the VDF graphs of deliberately insecure web applications. These metrics include 3 centrality metrics, 2 structural metrics, PageRank, in-degree, out-degree, and cross-clique connectivity. We present simulations that find that out-degree, betweenness centrality, in-eigenvector centrality, and cross-clique connectivity were found to be associated with files exhibiting high vulnerability traffic, making them refactoring candidates where input sanitization may have been missed. Meanwhile, out-eigenvector centrality, PageRank, and in-degree were found to be associated with nodes enabling vulnerability flow and sinks, but not necessarily where input validation should be placed. This is a novel method to automatically provide development teams an evidence-based prioritized list of files to embed security controls into, informed by vulnerability propagation patterns in the application architecture.**

***Index Terms*—SAST, vulnerability data flow, vulnerability remediation, graph theory, taint analysis**

I. Introduction

Eight of the top ten data breaches of 2023 were related to application attack surfaces [1]. This demands robust application-layer countermeasures, including active Static Application Security Testing (SAST) scanning. SAST performs analyses [2] including:

1) *Data flow analysis*: Follows the path of the data flow
2) *Control flow analysis*: Compares the application control flow on execution with known-secure code flow patterns
3) *Structural analysis*: Examines language-specific code structures for inconsistencies with best practices
4) *Semantic analysis*: Performs a simple search looking for known-insecure strings in the code base
5) *Configuration analysis*: Checks application configuration files against security best practices

Our research focuses on (1) and (2), as these techniques lend themselves readily to graphical visualization. We begin by noticing that SAST tools identify data flow paths (e.g., function calls) where tainted data passes through without security controls (e.g., input validation and sanitization). At present, flow data can be gathered from the UI of some major SAST platforms and is used by development teams to remediate vulnerabilities [3]. The potential for aggregating this data across vulnerabilities and running analytics on it remains untapped and is overlooked due to the cumbersome length of individual VDF data, which is typically excluded from standard reports [4].

Our research involves gathering and graphing tainted data paths, which will be referred to as vulnerability data flows (VDFs). We define a VDF as the propagation of an untrusted value from a source collection point in code to a destination or intermediate destination in code. When VDFs are graphed, nodes in the resulting graph signify files through which tainted data passes, while graph edges refer to flowing tainted data. This tainted data is directional and the resulting graph is a directed graph (digraph), which means analysis can be carried out to identify features like feedback loops, temporal sequence, and flow imbalance.

The graph serves two productive uses:

1) *Rapid consumption*: Rather than looking at VDFs for each vulnerability in isolation from each other through a SAST user interface, the graph reveals nodes shared by multiple vulnerabilities. As a result, the files contributing most to the vulnerability of the application will be identified. Sections of the applications that are most vulnerable will also be able to be identified.
2) *Generating insights using graph theory*: The graph of VDFs lends itself readily to measurement taking. We evaluate a variety of measurements in this paper, including centrality and other graph theory metrics, to determine their significance in an application security context.

The contribution of this research is the demonstration that the application of graph theory on application security results, specifically a constructed VDF graph, is meaningful for development teams looking to prioritize remediation.

This paper focuses on analyzing the VDF graph to optimize vulnerability reduction while minimizing the number of code commits. We test this experimentally by using graph theory metrics to identify files (i.e. graph nodes) involved in a maximum number of tainted data flows (i.e. paths in the graph) to retrieve a prioritized list of remediation and refactoring candidates. The remainder of this paper discusses existing work in the area and our contribution, followed by a discussion of our simulation set-up, results, and follow-on work. We demonstrate the insight that graph theory metrics can provide development teams so they can leverage vulnerability propagation patterns in application architectures when prioritizing tasks.

II. Related Works

While a study of the utility of graph theory metrics on VDF graphs has not been conducted to the best of our knowledge, there are a variety of approaches to analyze vulnerability data flows. Static taint analysis is one such approach and which tracks taints at the variable-level in code [5]. Much static taint analysis literature proposes domain-specific tools and methodologies such as Tripp et al.'s TAJ for Java [6] and Arzt et al.'s Flowdroid for Android [7], with the latter constructing a Taint Value Graph (TVG). Some modern approaches to taint analysis employ deep learning, with work including Niu et al's approach for IoT [8] and Chow et al.'s Fluffy [9] on top of GitHub's CodeQL analysis framework [10].

Data flow analysis is another avenue to analyze vulnerability data flows. Treating programmatic data flows using formal theory was expounded by F.E. Allen and J. Cocke from the IBM Thomas J. Watson Research Center [11] in 1976. This paper laid out formal definitions for nodes and edges of data flow graphs. Researchers from the University of Colorado Boulder then built on IBM's work in the same year to perform anomaly calculations on the data flow graph [12]. The body of literature continues to treat data flow graphs as directed and accessible by graph theory. With the advent of SAST, Checkmarx has published work on hierarchical data-flow graphs [13] in 2023.

Moreover, some work has commented on the time-consuming nature of SAST output consumption and proposed visualizations via an interactive dashboard [14]. This visualization data draws from commits rather than data flows. Meanwhile, there has been a body of research establishing the usefulness of stack trace data to estimate the attack surface [15] and to locate files responsible for specific vulnerabilities [16].

The area of software vulnerability assessment and prioritization has also been the subject of many surveys and reviews, including by Khan et al. (rule-based methods) [17], Kritikos et al. (static analysis) [18], Dissanayake et al. (socio-technical aspects) [19], and Le et al. (meta-survey) [20]. These surveys show studies of data-driven techniques (including multi-layer perceptron, random forest, and linear SVM) to analyze data sources including ExploitDB [21], NVD [22], dark web forums and markets [23], and other open-source repositories [24], [25].

This paper contributes to the body of knowledge by offering graph theory analysis methods for vulnerability data flow graphs, with the aim of prioritizing vulnerable files to remediate and to generate intuitive visualizations.

III. Solution Design and Implementation Details

VDFs are obtained with a SAST tool. VDFs are then taken from a Veracode API. The user can run VIVID locally on their machine after putting in the Veracode API key. Veracode expects VDF data in X format. VIVID can be used with existing SAST tools. VIVID is a collection of scripts that can be executed over a command line interface.

IV. Experiment Setup

When choosing metrics to analyze the VDF graph, it was appealing to include both local metrics such as degree and global metrics such as centrality metrics. This would allow insights into both direct and indirect relationships and vulnerability flow within the application architecture.

Nine metrics were chosen for analysis. These include 3 centrality metrics, 2 graph structural metrics, in-degree, out-degree, PageRank, and cross-clique connectivity. Refer to Table 1 for a comprehensive list of these metrics and their relevance in an application security context.

WebGoat (version 2023.8) and VeraDemo (version 2.1.1) were chosen as the application targets, both of which are deliberately insecure Java applications. The former is maintained by OWASP while the latter is maintained by Veracode, making them common targets for application security testing. Vulnerability data flows were then pulled from the API of a major SAST platform, from which the VDF graph was constructed.

After constructing the graph using Gephi [26] and GraphVis [27], an R script was developed and used to analyze the graphs against the 9 chosen metrics. Results and analysis discussion for each metric are presented below.

The VeraDemo VDF graph is presented in Figure 1.

The VDF graph of WebGoat is shown below. The graph includes many vulnerability islands in the architecture, implying that most vulnerabilities are not data-flow related or that the data flow is restricted to a single file. It is also seen that the large blue node in the middle corresponds to WebGoatUser.java, which contains constructors and serves up information such as the user role, username, or password on request.

A manual analysis was conducted on WebGoat code, finding that WebGoatUser employs a Model-View-Controller (MVC) architecture where any data validation is typically done as soon as the data is received, which is in the controller file. In other words, the file flagged as vulnerable by the SAST tool may not necessarily be the file where security controls such as validation or sanitization should be introduced.

To measure the success of the metrics in analyzing the VDF graph, we recall our objective to pinpoint files through which

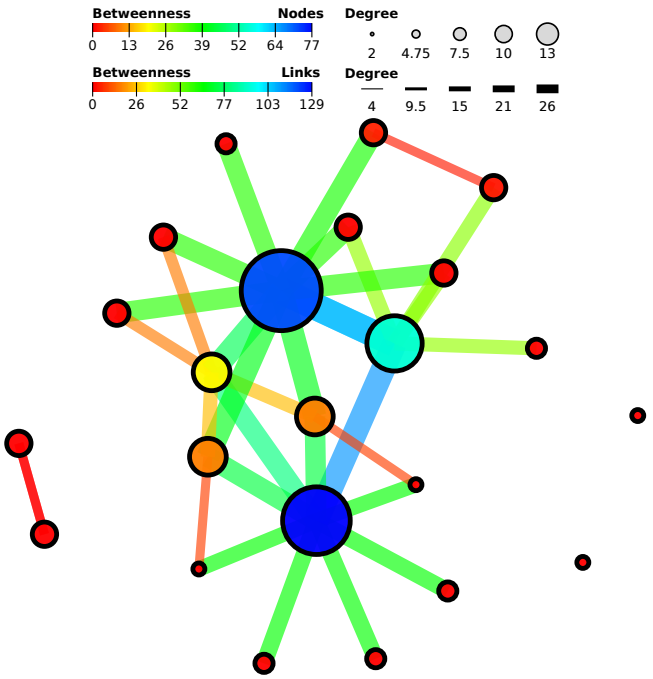

Fig. 1: VDF visualization of VeraDemo

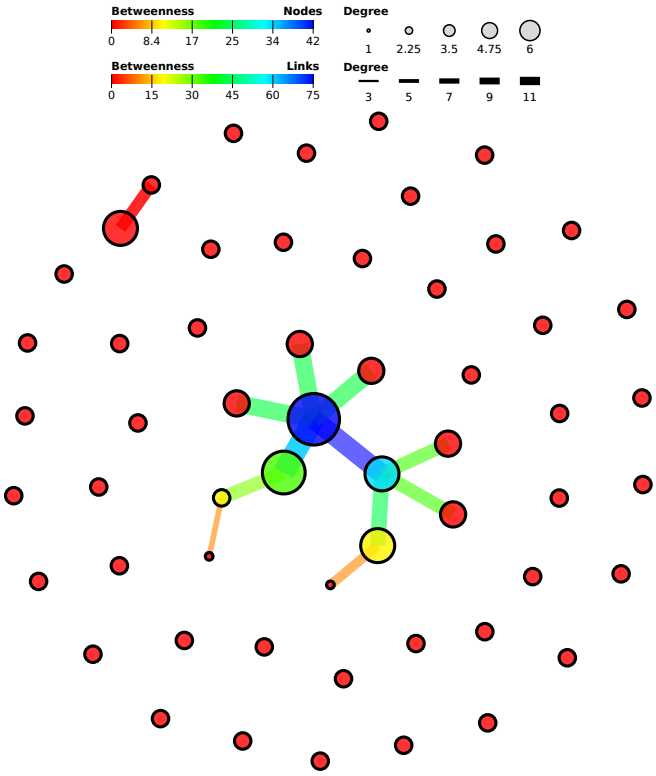

Fig. 2: VDF graph for WebGoat v2023.8

tainted data frequently traverses, enabling development teams to prioritize the integration of input validation, sanitization, and other security metrics. In so doing, we enumerated a list of 5 files of interest in WebGoat and VeraDemo from a manual code review and measure the success of the metrics in terms of the measure's capture of these files of interest.

All scripts are included in the referenced GitHub repository for reproduction.

V. Results and Discussion

Simulations on 2 deliberately insecure web applications were run and whose results are shown in the below radar graphs, where metrics are shown around the circumference of the graph. The graph shows how each of the 5 files of interest were ranked by the respective metric, where higher-ranked files are closer to the centre of the graph.

Figure 3 shows the WebGoat results, where in-degree and PageRank fail to capture several files of interest and out-eigencentrality captured some files of interest in their top-ranked files. Cross-clique connectivity, betweenness centrality, out-eigencentrality, and in-eigencentrality captured all files of interest in their top 5 rankings.

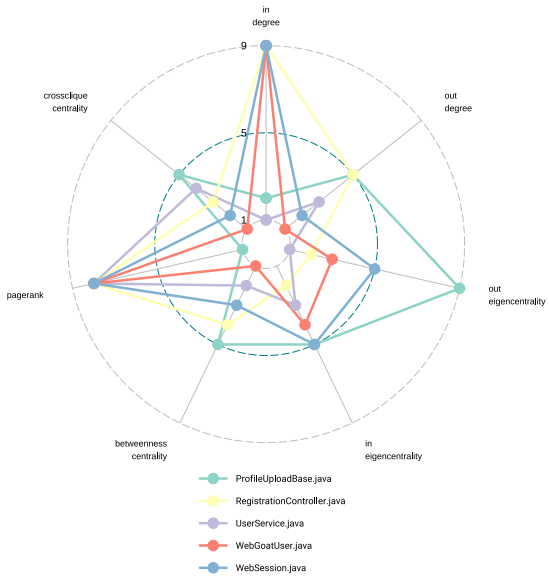

Fig. 3: Simulation results for WebGoat

Results from the VeraDemo simulation are shown in Figure 4. The results here echo WebGoat results in that PageRank fails to capture key of interest and that out-eigencentrality captures only some files of interest in its top 10. Note that the axis is re-scaled for VeraDemo to account for a larger number of nodes in the VDF graph.

A. Betweenness Centrality

Betweenness centrality is commonly used in network analysis. While existing tools like Sonargraph validate the entire

Measure	Definition	Significance in an Application Security Context
Out- and In-Eigenvector Centrality	$$\mathbf{v}_i = \frac{1}{\lambda}\sum_{j=1}^{N} A_{ij}\mathbf{v}_j$$ denotes the out-eigenvector centrality of file i, where A_{ij} indicates a tainted data path from file i to file j, and λ quantifies the maximum potential "traffic" or exerted influence in tainted data path network A_{ji} is used for in-eigenvector centrality.	Identifies files that propagate vulnerability chains and contribute to the impact magnitude of vulnerability spread. Pathways of nodes with high eigenvector centrality provide a clear picture of the predominant pathways of influence across the network. Development teams may choose to prioritize the remediation of vulnerabilities for which data passes through files with high eigenvector centralities whose compromise would entail a significant blast radius.
Substructure Entropy	$$H(v) = -\sum_{u \in V} p(u\|v)\log p(u\|v)$$ where $p(u\|v)$ is the probability of tainted data flowing from file v to file u.	Unusual occurrence, suggesting the relative level of refactoring effort required to remediate vulnerabilities.
Modularity	$$Q = \frac{1}{2\|E\|}\sum_{vw}\left[A_{vw} - \frac{\deg(v)\deg(w)}{2\|E\|}\right]\delta(c_v, c_w)$$ where $\|E\|$ is the number of tainted data paths and δ being the Kronecker delta function, checking membership.	Number of communities, indicating clustered vulnerabilities or inter-related security issues.
Betweenness Centrality	$$C_B(v) = \sum_{s \neq v \neq t}\frac{\sigma_{st}(v)}{\sigma_{st}}$$ where σ_{st} is total number of shortest tainted paths from file s to file t and $\sigma_{st}(v)$ denotes paths through file v.	Identifies bridges and bottlenecks in the application flow where vulnerabilities can have a higher spread or likelihood. A file with high betweenness centrality indicates that it is frequently encountered during the most efficient (shortest) routes that tainted data might take as it propagates through the system.
PageRank	$$PR(u) = \frac{1-d}{N} + d\sum_{v \in B_u}\frac{PR(v)}{L(v)}$$ with d as damping (˜0.85), N total files, B_u files sending tainted data to file u, and $L(v)$ files receiving tainted data from file v.	Identifies common sinks in vulnerability data flows. Optionally, PageRank weights may be assigned from the vulnerability's severity.
In-Degree	$$D^-(v) = \|(u,v) \mid (u,v) \in E\|$$ where u and v are files and E is the edge set.	Identifies functionally critical application areas where input validation is imperative. These include I/O interfaces, centralized databases, API endpoints, shared utilities, middleware, caches, inter-process buffers, and temp storage. High in-degree indicates essential roles, numerous interactions, and potential failure points.
Out-Degree	$$D^+(v) = \|(v,u) \mid (v,u) \in E\|$$ where u and v are files and E is the edge set.	Identifies vulnerablity disseminators. These include I/O outputs, database queries, API responses, library usages, middleware dispatches, cache updates, inter-process signals, and temp transfers. High out-degree suggests broad impacts, multiple dependencies, and influence spread.
Cross-Clique Connectivity	$X(v)$ is the number of cliques to which node v belongs.	Identifies highly cross-connected nodes, showing a tight coupling of vulnerabilities and node involvement in numerous vulnerability clusters.

TABLE I: Description of graph theory metrics in an application security setting

application architecture by constructing module dependency graphs and cyclicity graphs [28], there are no tools to model application vulnerabilities in the context of application architecture.

Armed with the concept of bridges in its mathematical definition, using betweenness centrality directly aligns with the stated objective of identifying files where tainted data frequently passes through. By focusing remediation on these nodes which correspond to bridges or chokepoints, development teams can address a disproportionately larger number of vulnerabilities, minimizing code commits while maximizing vulnerability reduction.

In the simulation of WebGoat, betweenness centrality determined the WebGoatUser.java to be the node with the highest betweenness centrality of 21. This node is seen in the centre of the graph, shown in Figure 3. Its surrounding nodes at betweenness centrality 16, 12, and 9. All remaining nodes have a betweenness centrality of 0.

The analysis shows that the WebGoatUser.java, which is the node with the highest degree (6) in the graph, is called and so accessible to its neighboring nodes. Placing a generic input validation function here would be a quick win as the function can be called by all its neighboring nodes and so the fix would mitigate many vulnerabilities. The simulation on VeraDemo confirms that betweenness centrality identifies high-value targets for input validation.

Simulation results for VeraDemo seen in 4 show that the BlabController.java has the highest betweenness centrality. Identifying a controller as a remediation target in an MVC application is a good sign. Interestingly, BlabController.java had the highest betweenness centrality whereas all other centrality metrics ranked UserController.java as the highest in their respective measure. This suggests that betweenness centrality provides key information on vulnerability remediation targets differentiated from other centrality metrics and which should be taken into account in a weighted formula that outputs a

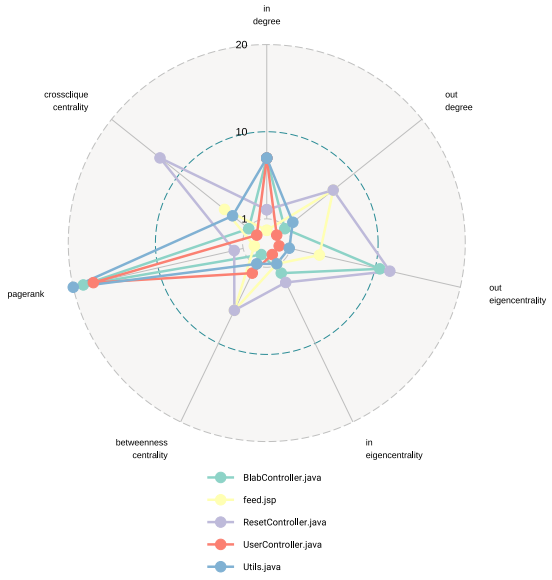

Fig. 4: Simulation results for VeraDemo

prioritized list of files to remediate.

B. Eigenvector Centrality

In the realm of networks, some nodes are not influential merely because they have many connections; they are influential because they connect to other influential nodes. The idea behind eigenvector centrality is to quantify this recursive notion of influence.

Nodes or sections of the VDF graph with high eigenvector centrality scores would be flagged as high-risk due to their potential cascading impact on the larger system. As a result, this may lead to an automatic generation of refactoring candidates.

Results on the WebGoat VDF show that 12 nodes have a non-zero out-eigenvector centrality. UserService.java has the highest out-eigenvector centrality (that is, of value 1), while the RegistrationController.java has an out-eigenvector centrality of 0.79. In-eigenvector centrality captures files including UserForm.java and ProfileUploadRetrieval.java in its top 5 rankings which are useful places to add in input validation.

In-eigencentrality out-performs out-eigencentrality for the VeraDemo simulation as the former picks up on 4 controller files in its top 5 rankings. Recalling that controllers are files of interest to insert security controls, this metric should be weighted higher when developing a formula to prioritize refactoring and insertion of security controls.

C. Modularity and Substructure Entropy

A modularity analysis was also carried out to visually represent clusters within the VDF graph. The cluster_walktrap was used as a modularity measure since it is appropriate for directed graphs.

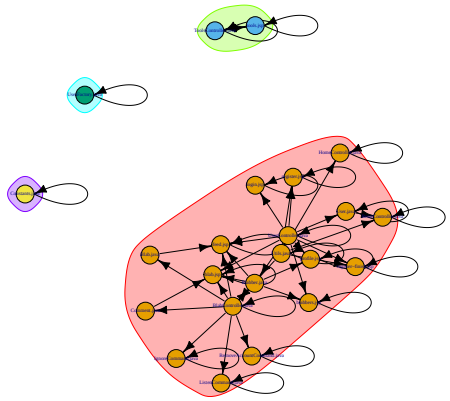

Fig. 5: Modularity plot for WebGoat v2023.8

Figure 3 shows the modularity plot of WebGoat. The large section of entropy 2.2 can be seen in the red background while the section of entropy 1 and 2 nodes can be seen on its upper left. The remainder of the nodes were not of sufficient degree to be included in the plot.

To be useful for development teams in prioritizing segments of the application, entropy calculations can be made on each of these vulnerability islands found in VDF graph.

Substructure entropy can be used for this purpose and offers immediate use cases for identifying refactoring candidates and third-party security contexts. It is a measure that captures the unpredictability or randomness of substructures within the VDF graph.

A higher entropy value suggests a diverse vulnerability propagation pattern, whereas a lower value might indicate recurring vulnerability patterns. This entropy measure provides a lens through which we can better understand the intricacy and patterns of vulnerabilities in applications. We enumerate two use cases for this measure:

1. Identifying refactoring candidates: Acquirers can leverage substructure entropy on the VDF graph to quantify and compare the refactoring effort required in different regions of the application code. Segments of the target software asset's code showing higher entropy in their VDF graph merit special attention, as they emerge as prime candidates for refactoring.

Sections of target software asset code with high substructure entropy represent potential risks and liabilities associated with the target's software asset. This is because these high-entropy code sections hinder the software's security maintainability and security resilience.

2. Third-party dependencies & External dependencies: 96% of codebases surveyed contain open-source software, according to Synopsys' 2023 Report on Open Source Security and Risk Analysis. More jarring is that 76% of code by volume was open-source [29]. Software assets not only integrate but largely lie on top of third-party libraries and dependencies. Given

the complexity of how first-party (1P) and third-party (3P) code intermingle and are tightly woven, upgrading outdated 3P software may require refactoring efforts. So, estimating the size of refactoring efforts involving 3P is of interest to development teams and the business.

At the time of writing, the authors were not able to find a tool that quantifies refactoring efforts involved in upgrading 3P libraries and packages to a secure version. Our suggested approach for this is to create a vulnerability data flow (VDF) graph on application code and rank software sections based on their entropy. Sections with higher entropy scores would need more intensive refactoring efforts.

When calculating substructure entropy using the R script provided in the GitHub repository, analysis minimizes noise by excluding nodes of degree two which are self-connections (loop) and not connected to any other node. In the case of WebGoat, there are 2 connected segments of entropy 2.2 and 1, which house 12 nodes and 2 nodes respectively. Similarly in VeraDemo, the substructure entropy analysis identifies 2 vulnerability islands, housing 2 and 19 nodes. The entropy analysis reveals an entropy of 2.47 for the 19-node segment.

Our analysis finds that these entropy values should be normalized so the segment's entropy can be interpreted as low, moderate, high, or very high. As such, we divide the found entropy by the maximum entropy given the degrees in the segment. The upper bounds for low, moderate, and high entropy were set as 0.25, 0.5, and 0.75 respectively.

The entropy values support the case that a higher remediation effort is needed for the 2.2 entropy section of 12 nodes. Moreover, the separation of the 2 substructures in WebGoat are a result of them being located in different folder paths.

D. Cross-Clique Connectivity

Cross-clique connectivity measures the propagation of information or disease in a graph [30], making it well-suited to VDF graph analysis. The cross-clique connectivity R package was used. This measure performed very well in the WebGoat simulation, as it captured 5 out of 5 (100%) files of interest. Cross-clique connectivity's performance for VeraDemo was similar to the other centrality metrics, indicating a consensus on the key files of interest.

E. In-Degree and Out-Degree

In and out-degree inform the extent of the in-flow and out-flow of tainted data flow respectively. Calculating files with the highest out-degree is equivalent to finding the list of files outputting the highest volumes of tainted data. This is important for developers to know where to place security controls including input validation.

In regards to the performance of the out-degree measure shown in Figures 3 and 4, out-degree alone was able to capture 3 of 5 (60%) major files of interest and was out-performed by centrality metrics such as cross-clique centrality which have a built-in understanding of influence in the larger graph.

In-degree ranked only 2 of 5 (40%) files of interest in its top 5 for WebGoat; however, it is a useful indicator of common sinks across multiple tainted data flow chains. Developers can insert post-operation checks or data integrity checks in files showing high in-degree. Nodes of high in-degree values should be provided as informational findings for development teams to insert these checks.

In-degree and out-degree performed identically for VeraDemo where the former ranked 2 of 5 (40%) files of interest in its top 5 while the latter captured (60%) in its top 5.

VI. Conclusion

Our paper demonstrates the fruitfulness of applying graph theory metrics to the VDF graph of an application. By harnessing vulnerability data flows (VDFs) and graphing them, we offer already data-saturated security departments an intuitive method to visualize application architecture.

We have shown that metrics including centrality metrics like cross-clique centrality can be applied out-of-the-box to VDF graphs, offering insight into prioritizing vulnerability remediations. Further, prioritization lists can then be generated automatically with further informational findings generated by in-degree file rankings.

Moreover, we clarified how centrality metrics and other metrics shed light on vulnerability contexts. Namely, the metrics capture the interplay between vulnerabilities, as well as propagation patterns of tainted data, flagging specific files for attention. From this, we infer that remediation across a small subset of files can significantly curtail vulnerability spread. Based on our results, we advocate for development teams to hone in on files marked by high scores in these metrics when planning remediation priorities.

A. Limitations & Future Work

Minimizing the number of code commits needed to maximize vulnerability reduction is but one of many avenues of analysis opened up by the use of graph theory on VDF graphs. A promising area of future work we are exploring is a points-based system, taking into account vulnerability severity. For example, higher points (i.e. priority) would be given to files where remediating one vulnerability (e.g., directory traversal) would mitigate multiple vulnerabilities, like command injection and XSS.

There also exists potential in evaluating additional metrics and identifying synergies between metrics with the goal of developing an index or combination of weighted centrality metrics. To this end, node colouring or overlay can be used to infuse criticality data into the graph and make the weighted formula of prioritized files to remediate sensitive to vulnerability criticality.

In the way of experiential learning, we have included in our GitHub repository a script enabling transformed VDF data to be viewed in a Virtual Reality (VR) simulation, allowing analysts to explore and interact with static vulnerability data in immersive environments.

Looking forward, the vulnerability graph may be more granular, where each node is a function within a file rather than the file itself. This would allow the developer to be

suggested a specific function where to put a input validator and sanitizer. A consideration on this approach is that many SAST vulnerabilities do not have a network effect; that is, they are contained within the same file, such as an insecure cipher suite vulnerability.

A noticeable gap exists in prioritizing remediation based on vulnerability severity. We look to examine the effectiveness of graph theory on graphs where weights correspond vulnerability severity and seek to publish our results in a follow-up study.

REFERENCES

[1] CrowdStrike, "State of Application Security Report," 2024. [Online]. Available: https://go.crowdstrike.com/rs/281-OBQ-266/images/report-2024-state-of-app-security-report.pdf?version=0.

[2] A. Bychkov, Artem. (2022). Know Your Tools: Quirks And Flaws, Integrating SAST Into Your Pipeline [Online]. Available: https://sec4dev.io/assets/uploads/slides/sec4dev2021_Know-your-Tools.pdf.

[3] M. Rao, "Why SAST tools aren't glorified GREP," Synopsys. Accessed: March 3, 2024. [Online]. Available: https://www.synopsys.com/content/dam/synopsys/sig-assets/ebooks/are-sast-tools-glorified-grep.pdf.

[4] J. Yang, L. Tan, J. Peyton and K. A Duer, "Towards Better Utilizing Static Application Security Testing," *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, Montreal, QC, Canada, 2019, pp. 51-60, doi: 10.1109/ICSE-SEIP.2019.00014.

[5] M. Velez. Foundations of Software Engineering - Taint Analysis [Online]. Available: https://www.cs.cmu.edu/~ckaestne/15313/2018/20181023-taint-analysis.pdf.

[6] T. Omar, M. Pistoia, S.J. Fink, M. Sridharan, O. Weisman, "TAJ: effective taint analysis of web applications," *ACM Sigplan Notices*., vol. 44, no. 6, pp. 87-97, Jun. 2009.

[7] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM Sigplan Notices*., vol. 49, no. 6, pp. 259-269, Jun. 2014.

[8] W. Niu, X. Zhang, X. Du, L. Zhao, R. Cao, and M. Guizani, "A deep learning based static taint analysis approach for IoT software vulnerability location," *Measurement*, vol. 152, p. 107139, Feb. 2020, doi: 10.1016/j.measurement.2019.107139.

[9] Y. W. Chow, M. Schäfer, and M. Pradel, "Beware of the Unexpected: Bimodal Taint Analysis," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 211-222, doi: 10.1145/3597926.3598050.

[10] GitHub. "CodeQL." GitHub.com https://codeql.github.com/ (accessed Dec. 12, 2023).

[11] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Communications of the ACM*, vol. 19, no. 3, p. 137, Mar. 1976. doi: 10.1145/360018.360025.

[12] L. D. Fosdick and L. J. Osterweil, "Data Flow Analysis in Software Reliability," *ACM Computing Surveys*, vol. 8, no. 3, pp. 305–330, Sep. 1976, doi: 10.1145/356674.356676.

[13] J. Pereira, V. Vieira, A. Simoes, "Hierarchical Data-Flow Graphs," *12th Symposium on Languages, Applications and Technologies (SLATE 2023)*, Aug. 2023, doi: 10.4230/OASIcs.SLATE.2023.11.

[14] A. Schreiber, T. Sonnekalb and L. v. Kurnatowski, "Towards Visual Analytics Dashboards for Provenance-driven Static Application Security Testing," *2021 IEEE Symposium on Visualization for Cyber Security (VizSec)*, New Orleans, LA, USA, 2021, pp. 42-46, doi: 10.1109/VizSec53666.2021.00010.

[15] C. Theisen and L. Williams, "Stack traces reveal attack surfaces," *Elsevier eBooks*, pp. 73–76, Jan. 2016, doi: 10.1016/b978-0-12-804206-9.00014-3.

[16] A. Iyer and L. M. Liebrock, "Vulnerability scanning for buffer overflow," *International Conference on Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004.*, Las Vegas, NV, USA, 2004, pp. 116-117 Vol.2, doi: 10.1109/ITCC.2004.1286600.

[17] S. A. Khan and S. Parkinson, "Review into State of the Art of Vulnerability Assessment using Artificial Intelligence," *Guide to Vulnerability Analysis for Computer Networks and Systems*, pp. 3–32, Oct. 2018, doi: https://doi.org/10.1007/978-3-319-92624-7_1.

[18] K. Kritikos, K. Magoutis, M. Papoutsakis, and S. Ioannidis, "A survey on vulnerability assessment tools and databases for cloud-based web applications," *Array*, vol. 3–4, p. 100011, Sep. 2019, doi: 10.1016/j.array.2019.100011.

[19] N. Dissanayake, A. Jayatilaka, M. Zahedi, and M. A. Babar, "Software security patch management—A systematic literature review of challenges, approaches, tools and practices," *Information and Software Technology*, vol. 144, p. 106771, Dec. 2021, doi: 10.1016/j.infsof.2021.106771.

[20] T. H. M. Le, H. Chen, and M. A. Babar, "A Survey on Data-driven Software Vulnerability Assessment and Prioritization," *ACM Computing Surveys*, Apr. 2022, doi: 10.1145/3529757.

[21] N. Bhatt, A. Anand, and V.S.S. Yadavalli, "Exploitability prediction of software vulnerabilities," *Quality and Reliability Engineering International*, vol. 37, no. 2, pp. 648–663, Sep. 2020, doi: 10.1002/qre.2754.

[22] B. L. Bullough, A. K. Yanchenko, C. Smith, and J. R. Zipkin, "Predicting Exploitation of Disclosed Software Vulnerabilities Using Open-source Data," *Proceedings of the 3rd ACM on International Workshop on Security And Privacy Analytics*, pp. 45-53, Mar. 2017, doi: 10.1145/3041008.3041009.

[23] M. Almukaynizi, E. Nunes, K. Dharaiya, M. Senguttuvan, J. Shakarian and P. Shakarian, "Proactive identification of exploits in the wild through vulnerability mentions online," *2017 International Conference on Cyber Conflict (CyCon U.S.)*, Washington, DC, USA, 2017, pp. 82-88, doi: 10.1109/CYCONUS.2017.8167501.

[24] C. Xiao, A. Sarabi, Y. Liu, B. Li, M. Liu, and T. Dumitras, "From patching delays to infection symptoms: Using risk profiles for an early discovery of vulnerabilities exploited in the wild," in *27th USENIX Security Symposium*, Aug. 2018, pp. 903-918.

[25] Y. Jiang, Y. Atif, "An approach to discover and assess vulnerability severity automatically in cyber-physical systems," in *13th International Conference on Security of Information and Networks*, Nov. 2020, pp. 1-8.

[26] *Gephi*. (2023), M. Bastian, E.R. Ibañez, M. Jacomy, C. Bartosiak, S. Heymann, J. Bilcke, P. McSweeney, A. Panisson, J. Subtil, H. Suzuki, M. Skurla, A. Patriarca, Accessed January 2023. [Online]. Available: https://gephi.github.io/.

[27] R.A. Rossi, N.K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, Jan. 2015, pp. 4292-4293.

[28] Hello2morrow. *Sonargraph Manual*. Accessed Sept. 2023. [Online] Available: https://eclipse.hello2morrow.com/doc/standalone/content/index.html.

[29] Synopsys. "Synopsys' 2023 Report on Open Source Security and Risk Analysis," Synopsys, 2023. Accessed: Oct. 2023. [Online]. Available: https://go.snyk.io/state-of-open-source-security-report-2023.html.

[30] M. R. Faghani and U. T. Nguyen, "A Study of XSS Worm Propagation and Detection Mechanisms in Online Social Networks," in *IEEE Transactions on Information Forensics and Security*, vol. 8, no. 11, pp. 1815-1826, Nov. 2013, doi: 10.1109/TIFS.2013.2280884.