CROSS DOMAIN VULNERABILITY DETECTION USING GRAPH CONTRASTIVE LEARNING

Mahmoud Zamani, Saquib Irtiza, Shamila C. Wickramasuriya, Latifur Khan & Kevin W. Hamlen
Department of Computer Science
The University of Texas at Dallas
Richardson, TX 75080, USA
{mahmoud.zamani, saquib.irtiza, scw130030, lkhan, hamlen}@utdallas.edu

ABSTRACT

To overcome the difficulty of finding good-quality labeled data in domains such as vulnerability detection, Self–Supervised Learning (SSL) methods such as Contrastive Learning (CL) algorithms were developed. We evaluate the performance of one such state-of-the-art CL method, GraphCL, that trains on our graph dataset generated from code repositories of six widely used C/C++ applications. We also propose a custom graph type having a new structure that combines both code-level and binary-level CPG graphs. This is because, even though existing graph types such as AST, CFG and CPG are effective in detecting vulnerabilities in the source code, it is ineffective in detecting the ones that only occur in the binary-level. Hence, to detect those vulnerabilities, we propose a new graph type, Cross Domain Control Property Graph (CDCPG). We perform extensive experiments, using different augmentation techniques and loss functions to show that our custom graph type, CDCPG, performs better than other graph types in many scenarios.

1 INTRODUCTION

Automatic detection of software vulnerabilities is a major topic of research in cybersecurity because of the tendency of these vulnerabilities to cause significant damage to the system's confidentiality and availability. Especially with the rise in the number of public open-source repositories, the detection of vulnerabilities is more important now than ever. A vulnerable program contains flaws that allows malicious actors to exploit its contents and harm any hardware or software associated with it.

Due to the poor true positive and true negative rates of traditional methods (Li et al., 2019; Xu et al., 2017) and the cost of manually extracting feature for machine learning models (Nguyen & Tran, 2010; Neuhaus et al., 2007), researchers relied on automatic feature extraction using Deep Learning techniques (Li et al., 2018; 2021). These methods performed well but overlooked the control flow, data flow and syntactic structure of a program. Instead, they considered the programs as natural language problems. To overcome this issue, many works (Zhou et al., 2019; Chakraborty et al., 2021) started using graph learning methods on graphs like Control Property Graph (CPG) (Yamaguchi et al., 2014), Abstract Syntax Tree(AST) and Control Flow Graph (CFG) to incorporate syntactic and semantic information of the codes while learning the representations. But these methods all required source codes to generate the graphs which are often not publicly available. Instead, only binary (.bin) and executable (.exe) files are shared.

This motivated us to generate CPGs from binary files, which we call BIN, so that we can detect vulnerabilities even if the source code is unavailable. Also, during compilation, additional complexities such as abstraction of function and variable names, reordering of stack variables etc. are often introduced that might lead to new vulnerabilities only detectable at binary-level. That is where our new custom graph type called Cross Domain Control Property Graph (CDCPG), plays a crucial role. It is able to identify vulnerabilities both in the source code and its corresponding compiled binary code. Figure 3 in A.1 shows a sample. Also, most existing methods use supervised learning models which require a large volume of labeled data to train. Since it is very expensive to collect such a large dataset in this domain, we use state-of-the-art Self-Supervised Learning (SSL) method, particularly GraphCL, to evaluate our curated dataset. Our key contributions are as follows:

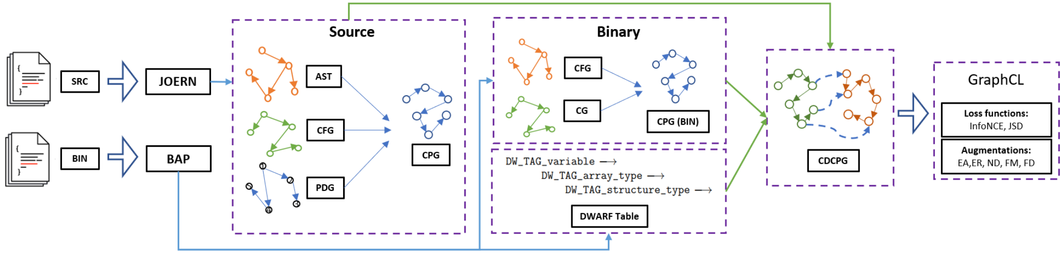

Figure 1: Data collection along with graph contrastive learning technique. More details in A.3.

- We developed a new graph type called CDCPG that combines both binary-level code and source-code level features. We collect graphs for six widely used open-source programs into a new curated dataset for vulnerability detection. Additionally, the dataset contains other graph types such as AST, CPG and CFG.
- We evaluated the performance of CDCPG on a contrastive learning model, GraphCL, to see whether it performs better than other graph types. We also use different loss functions and augmentation techniques to identify the best performing combination for this algorithm.

2 Data Collection and Proposed Method

Initially, we go over open-source software repositories of six different applications and collect functions from them that are marked as vulnerable according to Common Vulnerabilities and Exposures (CVE) (MITRE, 1999). We also collect the corresponding binary code for those functions. Then we use JOERN (Yamaguchi, 2021), a C/C++ analysis framework to generate AST, CFG and CPG graphs from the source level codes. We also generate BIN graphs from the corresponding binary code using a tool called Binary Analysis Platform (BAP) (Brumley et al., 2011). Once these graphs are ready, we combine them into CDCPG. Next, we use these graphs to train GraphCL. During the training phase, we use various loss functions (InfoNCE, Jensen-Shannon Divergence(JSD)) and augmentation techniques (Edge Addition (EA), Edge Removal (ER), Node Dropping (ND), Feature Masking (FM), Feature Dropping (FD)) to find for which combination of these properties the model performs the best. Figure 1 gives an overview of the pipeline. You can refer to A.3 for more details.

3 Evaluation

We use two different metrics for evaluation, F1-micro and F1-macro, to account for any data imbalance issue in the dataset. F1-micro gives equal weight to all the instances whereas F1-macro gives equal weight to each class. This allows both minority and majority class to contribute equally towards F1-macro. Our experiments are conducted on each application separately for all the graph types, loss functions and augmentation techniques mentioned in Section 2. Table 1 shows the result.

Table 1: F1-scores using ND augmentation and JSD loss function with GraphCL algorithm on all graphs. Results show that CDCPG outperforms other graph types for the TCPDump application.

Graph Types	AST	CFG	CPG	BIN	CDCPG
F1-Micro	0.85	0.63	0.63	0.63	**1.00**
F1-Macro	0.46	0.39	0.39	0.39	**1.00**

4 Conclusion

We proposed a new graph dataset called CDCPG (CPG and Binary Graph) to detect vulnerabilities in source and binary level. To train the model, we used contrastive learning technique with multiple graph augmentation methods and evaluated the result based on different loss functions.

URM STATEMENT

The authors acknowledge that at least one key author of this work meets the URM criteria of ICLR 2023 Tiny Papers Track.

REFERENCES

David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pp. 463–469. Springer, 2011.

Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering*, 2021.

Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 533–544, 2019.

Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2244–2258, 2021.

MITRE. Common vulnerabilities and exposures. `https://www.cve.org/`, 1999. Accessed: 2023-04-23.

Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*, pp. 529–540, 2007.

Viet Hung Nguyen and Le Minh Sang Tran. Predicting vulnerable software components with dependency graphs. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, pp. 1–8, 2010.

Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. Spain: security patch analysis for binaries towards understanding the pain and pills. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 462–472. IEEE, 2017.

Fabian Yamaguchi. Joern - the bug hunter's workbench. `https://docs.joern.io/home`, 2021. Accessed: 2023-04-23.

Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pp. 590–604. IEEE, 2014.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

A APPENDIX

A.1 SAMPLE OF CDCPG GRAPH

Figure 3 shows a sample CDCPG graph generated for the code snippet in 2. The paths labeled with blue and grey arrows in Figure 3 belong to the CPG generated from the source-level code whereas the path represented by black arrows are the ones that belong to the CPG of the binary-level code. The dotted purple lines show the mapping between the different components in the source and binary level CPG graphs.

```
void foo(){
    int x = source();
    if (x < MAX){
        int y = 2   x;
        sink(y);
    }
}
```

Figure 2: Code Snippet from which the CDCPG in Figure 3 is generated.

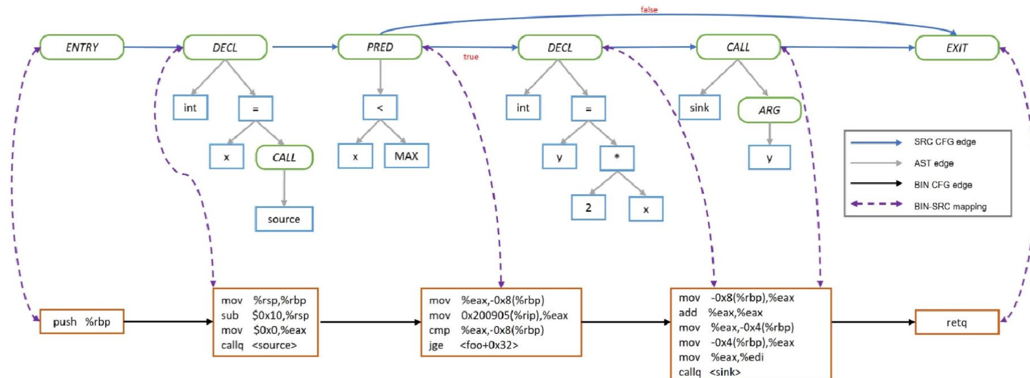

Figure 3: Cross-Domain Code Property Graph (CDCPG) generated from the snippet in Figure 2.

A.2 Additional Experiments

We conducted further evaluation on just CDCPG graphs using different augmentation techniques and loss functions to find the optimal parameters for GraphCL on the vulnerability detection task. Table 2 shows the scores averaged across all the six applications in the dataset. The highlighted scores show the ones that are the highest for each augmentation mode. It is evident that JSD is the better performing loss function because it gives the highest score for three out of the five augmentation techniques using both the metrics. Also, for JSD loss, the best performing augmentation technique is Node Dropping (ND) since it gives the highest scores for both F1-Micro and F1-Macro.

Table 2: Evaluation of different augmentations and loss functions using CDCPG for all applications

Modes	InfoNCE-F1Mi	InfoNCE-F1Ma	JSD-F1Mi	JSD-F1Ma
EA	0.852	0.522	**0.863**	**0.565**
ER	**0.843**	0.585	0.825	**0.600**
ND	0.738	0.522	**0.885**	**0.650**
FM	0.843	**0.522**	**0.883**	0.468
FD	**0.875**	**0.555**	0.768	0.425

A.3 Dataset Description

Our dataset contains graphs from six different applications that are widely popular in the open source community. We selected these applications based on the number of security issues reported against them and also depending on the importance of the applications. This means, applications dealing with critical security functionalities such as network security protocols, were also included in the dataset because having undetected vulnerabilities in these applications could jeopardise the security of the entire system. The description of each of the six applications have been included in Table 4. We collected all five types of graphs; AST, CFG, CPG, BIN and CDCPG for all the applications.

Table 3 shows a detailed description of the dataset including the number of vulnerable an non-vulnerable instances.

BAP generates CFGs from binary code which we then combine with Call Graphs (CG) to facilitate analysis between different functions. CGs are graphs that represent the calling relationship between different call events in a program. Next, binary attributes are extracted from this new combined graph to form the binary-level CPG graphs. Finally, to determine the relationship between executable and original source code, the Debug With Arbitrary Record Format (DWARF) table is generated during compilation that helps us to link different components of source-level graph with binary-level CPG graphs. Once the links are established, we obtain our CDCPG graphs.

Table 3: Dataset description. The last two columns report the number of vulnerable and non-vulnerable functions, respectively.

App Name	Graph Types	Edges	Nodes	Graphs	Vul.	Non-Vul.
LibPNG	CPG	1845382	82413	324	25	299
	CFG	3104607	12487			
	BIN	7861098	28248			
	AST	2628710	81777			
	CDCPG	8415974	116839			
LibTIFF	CPG	5125862	79009	438	33	405
	CFG	4498837	13142			
	BIN	9156127	47104			
	AST	3679494	78132			
	CDCPG	5746318	116839			
TCPDump	CPG	2074116	33461	100	15	85
	CFG	328240	6018	100	15	85
	BIN	607635	13679	100	15	85
	AST	6092203	57465	200	26	174
	CDCPG	4017677	48944	100	15	85
Sudo	CPG	5636108	41858	187	9	178
	CFG	740870	6213			
	BIN	2664002	25731			
	AST	4384746	41506			
	CDCPG	1617438	71463			
TinTin	CPG	8764300	44903	270	4	266
	CFG	1006857	6636			
	BIN	3747693	24376			
	AST	586926	44363			
	CDCPG	2361790	78414			
OpenSSH	CPG	5750707	70560	100	21	79
	CFG	732305	1003			
	BIN	1965966	36199			
	AST	4186296	70352			
	CDCPG	10937966	112250			

Table 4: List of target applications and their descriptions

Application	Description
Sudo	Delegates security privileges to other users or tasks
Poftpd	Ftp server with configurable features
Libtiff	Tagged Image File Format (TIFF) library
Libpng	Portable Network Graphics (PNG) library
Freetype	Renders text into bitmaps
TinTin	Console telnet client for online gaming
Tcpdump	Data network packet analyzer
OpenSSH	Secure networking utility for Secure Shell protocol