Statically Detecting Software Vulnerabilities using Deep Graph Neural Network

Xiao Cheng

1. Introduction

Modern software systems are often plagued with a wide variety of software vulnerabilities. Detecting and fixing these complicated, emerging and wide-ranging vulnerabilities are extremely hard. The number of vulnerabilities registered in the Common Vulnerabilities and Exposures (CVE) [1] has increased significantly during the past three years. Statistics show that there are 34,473 newly registered CVEs and 34,093 of them are above medium-level security from 2017.01.01 to 2019.07.20 according to NVD [1].

2. Challenges

Static bug detection, which approximates the runtime behavior of a program without running it, is the major way to pinpoint bugs at the early stage of software development cycle, thus reducing software maintenance cost. Traditional static analysis techniques (e.g., Clang static analyzer[2], Coverity[3], Fortify[4], Flawfinder[5], Infer[6], ITS4[7], RATS[8], Checkmarx[9] and SVF[10]) have shown their success in detecting well-defined memory corruption bugs.

Adapting the existing solutions for detecting a wide variety of emerging vulnerabilities has two major drawbacks. First, they rely on static analysis experts to define specific detection strategies for different types of vulnerabilities, which is labour-intensive and time-consuming. Second, the effectiveness of the pre-developed detection systems highly relies on the expertise of the analyzer developers and the knowledge of existing vulnerabilities. The emerging high-level vulnerabilities poses big challenges to existing bug detection approach, making it hard to extend the existing bug detectors.

3. Objectives

Existing solutions focus on detecting low-level memory vulnerabilities and failed to preserve comprehensive code features. This project aims to systematically investigate how to apply learning-based program analysis in detecting both high-level and low-level software vulnerabilities by leveraging the massive amount of available data.

Plenty of graphic structure like control-flow graph, data-flow graph could reflect both textual and code structured features of the source code. My plan is to first perform program analysis on source code to extract fine-grained but complicated semantic features, and then combine with graph neural networks to produce compact and low-dimensional representation.

4. Methodology

First, construct interprocedural CFG, VFG of program and adopt code slicing on them to build a graphic structure that could both capture control- and data-flow related semantic features of the source code. An interprocedural control-flow graph (CFG) of program is denoted as $G = (V, E)$, consisting of a set of vertices and a set of directed edges. Each node represents a basic block (i.e., a straight-line of code statements without conditional jumps) and each edge connects two nodes, signifying the control-flow or execution order between two basic blocks. VFG is a directed graph that captures its def-use chains conservatively.

Second, design an algorithm to combine CFG and VFG thus preserving both control- and data-flow semantic features. We then collect the graphs of both correct and vulnerable programs to train a bug prediction model using graph neural network (GNN), producing a structure-preserving code representation. For a vulnerable program, its graph represents program paths may trigger a bug. For a correct program, graph represents the safe programming patterns that unlikely trigger a bug. By learning the existing bug/safe patterns using recent advances in GNN, this framework supports precise bug prediction by pinpointing directly to the potentially vulnerable program paths (slices) in the source code.

5. Feasibility

My previous research presents a new deep-learning-based graph embedding approach to accurate detection of CFR(control-flow-related) vulnerabilities[11]. I am familiar with graph embedding tech used on my previous research and I will work on the proposed topic by applying it to detect both high-level and low-level vulnerabilities. The existing open-source tool SVF [10] (developed by Dr. Sui) can be a good base for my forthcoming research project. SVF is publicly available and hosted on Github. It is a fundamental program analysis infrastructure for program understanding and software bug detection. The tool and its related publications has been used and cited by many developers and researchers world-wide.

6. Schedule

Months 0–6
Summarize previous work and expand literature review;
Data collection and comparative study;
Months 7–12
Specify my research target and plan;
Collect methods that could aid my specific project;
Months 13–24
Propose idea and design the corresponding method;
Validate the proposed method;
Months 25–30
Write research papers while continue complete my proposed method;
Update my research progress into the research paper;
Ask external reviewers to review the paper;
Months 31–36
Summarize my research;
Write my PhD thesis;
Submit the PhD thesis to UTS graduate school and review.

References

[1] NATIONAL VULNERABILITY DATABASE, 2019. https://nvd.nist.gov/.
[2] 2019. Clang static analyzer. https://clang-analyzer.llvm.org/scan-build.html.
[3] 2019. Coverity. https://scan.coverity.com/.
[4] 2019. HP Fortify. https://www.hpfod.com/.
[5] 2019. Flawfinder. https://dwheeler.com/flawfinder/.
[6] 2019. Infer. https://fbinfer.com/.
[7] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. 2000. ITS4: a static vulnerability scanner for C and C++ code. In Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00). 257–267. https://doi.org/10.1109/ACSAC.2000.898880
[8] 2014. RATS. https://code.google.com/archive/p/rough-auditing-tool-forsecurity/.
[9] 2019. Checkmarx. https://www.checkmarx.com/.
[10] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. https://github.com/unsw-corg/SVF. In CC '16. 265–266.
[11] X. Cheng, H. Wang, J. Hua, M. Zhang, G. Xu, L. Yi, and Y. Sui. Static detection of control-flow-related vulnerabilities using graph embedding. In2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS), pages 41–50, Nov 20