Vulnerability Detection in Source Code via Subgraph Isomorphism Analysis

Douglas Takada, Freeman Trader, Tashi Stirewalt, and Assefaw Gebremedhin

ABSTRACT

Vulnerability detection is crucial for ensuring, the security, reliability, and privacy of software systems. By representing **source code as a property graph**, where nodes represent code elements and edges represent relationships between them, we are able to perform a comprehensive vulnerability assessment. With the source code represented as a graph, we can leverage graph-based algorithms, and machine-learning techniques to identify potential vulnerabilities. In this study, we introduce two things: a **novel graph representation** of source code based on the **Abstract Syntax Tree (AST)** and a novel approach to **estimating subgraph isomorphism**. We demonstrate the effectiveness of using graph analysis of source code for vulnerability detection. It indicates that such methods can be particularly valuable for **detecting well-understood and documented vulnerabilities**.

RELATED WORK

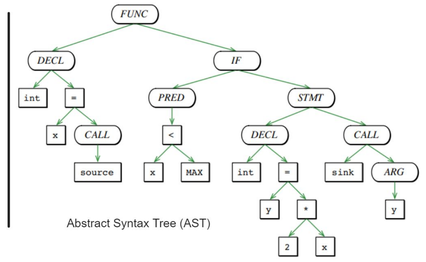
The research paper titled "Modeling and Discovering Vulnerabilities with Code Property Graphs" by Fabian Yamaguchi et al. [1] presents a **novel method for representing source code known as a code property graph**. This graph consolidates traditional program analysis concepts into a single data structure. While effective, this approach may face **scalability issues** when dealing with large code bases. In the paper "Abstracting Program Dependencies using the Method Dependence Graph" by Haipeng Cai et al. [2], a new dependence abstraction called the **method dependence graph (MDG)** is introduced. It **approximates the traditional fine-grained software dependence model** at the method level. "Source Code Vulnerability Detection: Combining Code Language Models and Code Property Graphs" by Ruitong Liu et al. [3] proposes a **unified model combining pre-trained code language models with code property graphs for vulnerability detection**. However, this method might **underestimate code semantics** and encounter challenges capturing long-distance contextual information. The work on "Source Code Vulnerability Detection Using Deep Learning" [4] focuses on **static security vulnerability detection** in C# code using **deep learning algorithms**. Lastly, "An Analytical Review of the Source Code Models for Exploit Analysis" [5] investigates a new approach for detecting vulnerabilities that can be **exploited in cyber attacks**.

ABSTRACT SYNTAX TREE

An **Abstract Syntax Tree (AST)** represents the **structure of a program** in a tree format. Each node in the tree represents a part of your code. For example, a function could be a parent node, and the statements inside that function could be its children. It's called **"abstract" because it simplifies the code** by leaving out some details like punctuation and brackets. ASTs are used in many areas of computer science, such as compilers and code analysis tools.

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

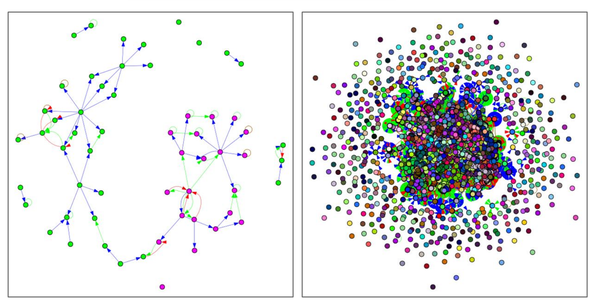
Source Code



Abstract Syntax Tree (AST)

GRAPH REPRESENTATION

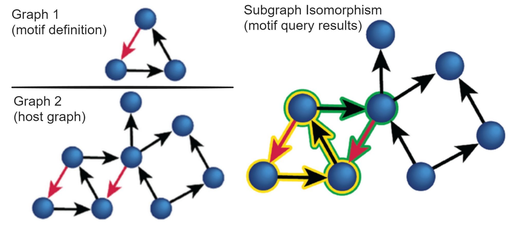
1. **Graph nodes = Classes/Functions**
2. **For each node:**
   a. **Get edges**
      i. **Child (node defined "under" it)**
      ii. **Call (node that it "calls")**
      iii. **Return (if node explicitly returns, then edge to match each 'Call' to it)**
   b. **Build node "profile" for each node**
      i. **Whether *CLASS* or *FUNCTION***
      ii. **Whether is *ASYNCHRONOUS***
      iii. **Whether explicitly *RETURNS***
      iv. **Number of *PARAMETERS***
      v. **Number of *CALLS***
      vi. **Number of *CHILDREN***



Extracted graphs from files of the 'Torch' package colored by file. Left is 2 arbitrary files, right is the entire package.

SUBGRAPH ISOMORPHISM (SI)

Subgraph isomorphism is a concept in **graph theory** that deals with the **structural relationship** between two graphs. Two graphs are considered **isomorphic** if there is a **one-to-one correspondence** between their vertices and edges that preserves the connectivity of the graphs. A graph G is a **subgraph isomorphic** to another graph H if a subgraph of H is isomorphic to G. This means there's a way to **map each vertex in G to a unique vertex in H** and each edge in G to a unique edge in H, **preserving connectivity**.



Graph 1: smaller graph that defines structure to look for. **Graph 2:** larger graph that defines structure to look over. **Results:** Set of subgraphs of Graph 2 that are isomorphic Graph 1.

SI ESTIMATION

Given **profile embeddings** for each node in the graph, we can **calculate distances** between nodes in Graph 1 and Graph 2. We then create a **ranking of node correspondence candidacy** for all pairs of nodes. Using these lists, we generate unique node-to-node correspondences and **evaluate their connectivity**. These checks can be **parallelized**, but we compute a **bounded solution** due to the lack of guaranteed results with large graphs.

Compute node profile distances

For each motif node profile C versus host node profile A.

C = [0, 0 , 0, 0, 2, 4] **A = [1, 0 , 1, 0, 0, 1]** **A = [0, 0 , 0, 0, 7, 16]**

i. 1 for each of first three not equal

A = [1, 0 , 1, 0, 0, 1] **A = [0, 0 , 0, 0, 7, 16]**

ii. Euclidean for last three

A = [1, 0 , 1, 3.60555…] **A = [0, 0 , 0, 13]**

iii. Normalize Euclideans (let max = 87.057…)

A = [1, 0 , 1, 0.04140…] **A = [0, 0 , 0, 0.14932…]**

iv. Matrix multiplication w/ weights (let weights = [0.5, 0.1, 0.1, 0.3])

A = 0.61242… **A = 0.04479…**

Therefore results "distances" are 0.61242… and 0.04479…

Node Correspondence Combination

Component nodes

0 1 2 3 4 5 6 7 8 9 10

Application candidate nodes

47 48 49 50 51 52 53 54 55 56 57

Connectivness Evaluation

A47 as C0: matches 100% each specified by component

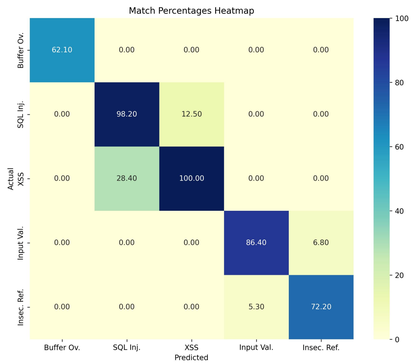
46,347:call
48,47:call → 1,0:call
47,48:return → 0,1:return
73,47:call
47,73:return

MITRE CWE & CVE

MITRE's Common Weakness Enumeration (CWE) is a community-developed list of common software weaknesses. **MITRE's Common Vulnerabilities and Exposures (CVE)** is a list of known security vulnerabilities.

RESULTS

We created a **motif** graph for **five classic vulnerability** types defined under CWE. Then, the same **vulnerability is inserted** somewhere in a **larger code file**, and host graphs are created. Each host graph is then **checked for the presence of each motif**, and only the best-found result above at least 5% match is reported.



REFERENCES

[1] Yamaguchi, Fabian . "Modeling and Discovering Vulnerabilities with Code Property Graphs." (2014).

[2] Cai, Haipeng. "Abstracting Program Dependencies using the Method Dependence Graph." (2015).

[3] Liu, Ruitong. "Source Code Vulnerability Detection: Combining Code Language Models and Code Property Graphs." (2024).

[4] Louati, Akram. "Source Code Vulnerability Detection Using Deep Learning for Industrial Applications". (2023).

[5] Fedorchenko, Elena. "An Analytical Review of the Source Code Models for Exploit Analysis." (2023).

ACKNOWLEDGEMENTS

This work is supported by funding for the VICEROY Northwest Institute for Cybersecurity Education and Research (CySER) provided by The Office of the Undersecretary of Defense for Research and Engineering, in collaboration with the Air Force Research Laboratory and Griffiss Institute.