Using Program Knowledge Graph to Uncover Software Vulnerabilities

Mengjie Xie[1], Tamjid Al Rahat[2], Wei Wang[2], Yuan Tian[2]

[1]*University of California, Irvine* [2]*University of California, Los Angeles*

01 INTRODUCTION

In the rapidly evolving landscape of security vulnerabilities, security analysts and researchers require novel approaches to establish connections between semantics of software vulnerabilities and their presence in real-world systems. The concept of a knowledge graph, initially introduced by Google, has found many applications in representing diverse forms of vulnerability information. However, its untapped potential for comprehensively capturing software defects and vulnerabilities within real systems has yet to be fully investigated.

In this poster, we propose a novel approach of ***Program Knowledge Graph*** that expands the vulnerability knowledge with program graphs (e.g., call graph, data flow, etc.) extracted from the implementation of security-critical systems. Further, leveraging prompt tuning with large language models (LLMs), we propose to automatically generate queries that assess the presence of specific vulnerabilities hidden underneath the complex systems. This methodology empowers developers to scrutinize and fortify their implementation against potential security risks at runtime. While existing work demands manual query construction, our approach seeks to alleviate this challenge by harnessing LLMs for query auto-generation.

02 SYSTEM OVERVIEW

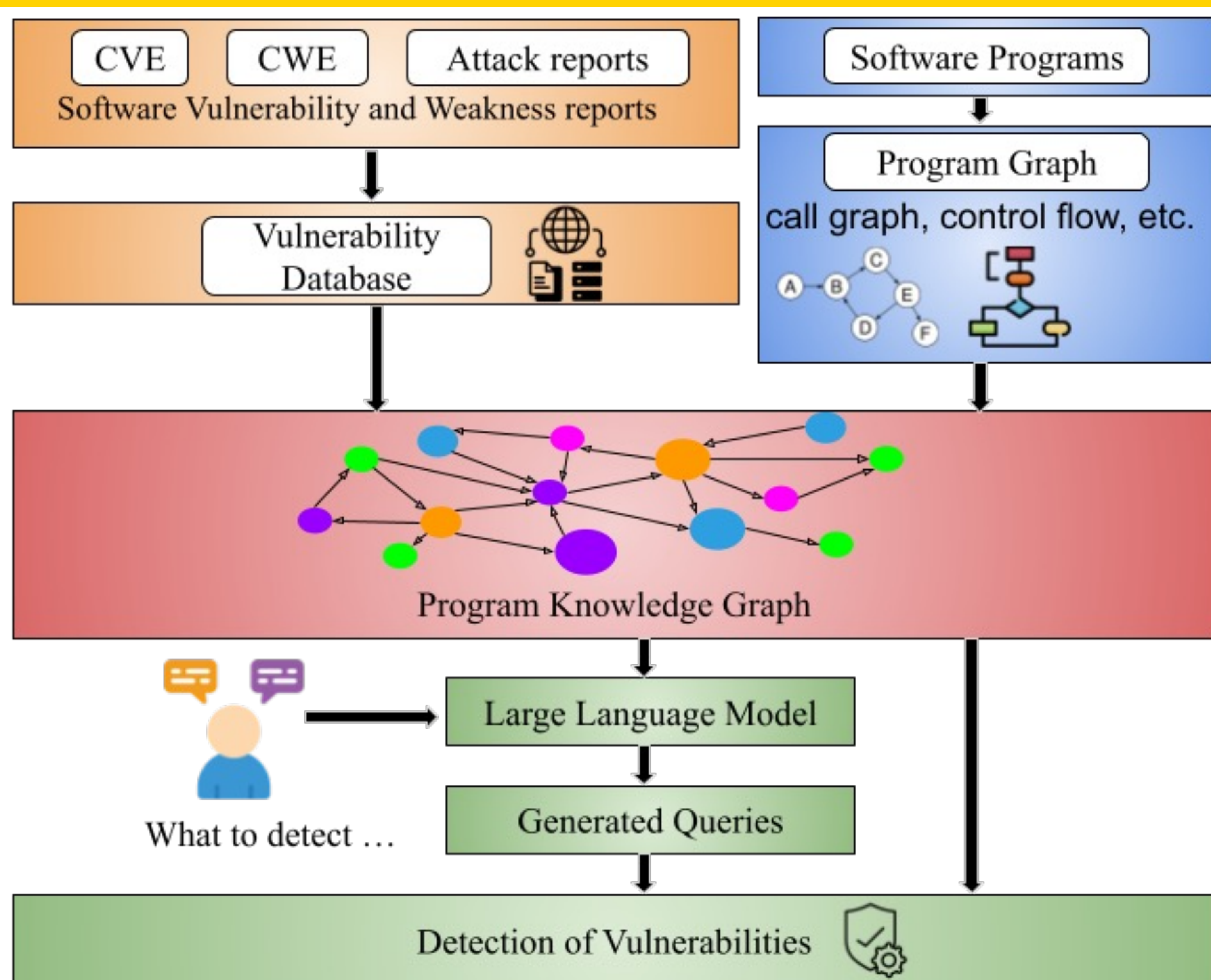
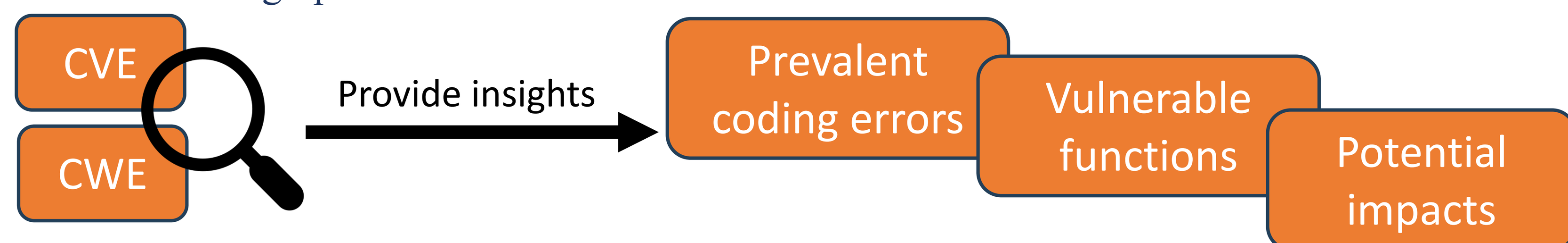


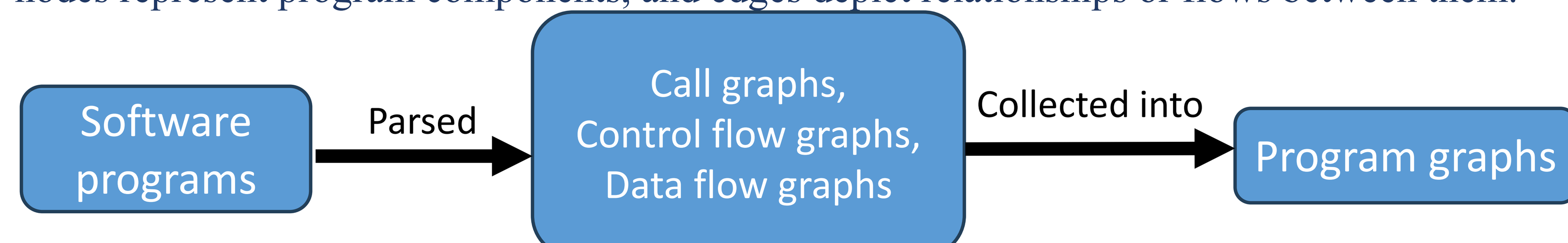
Figure 1: Schematic architecture of our *Program Knowledge Graph* to detect vulnerabilities in software programs.

Our proposed approach of using *program knowledge graph* to uncover software vulnerabilities encompasses a series of structured stages (Figure 1):

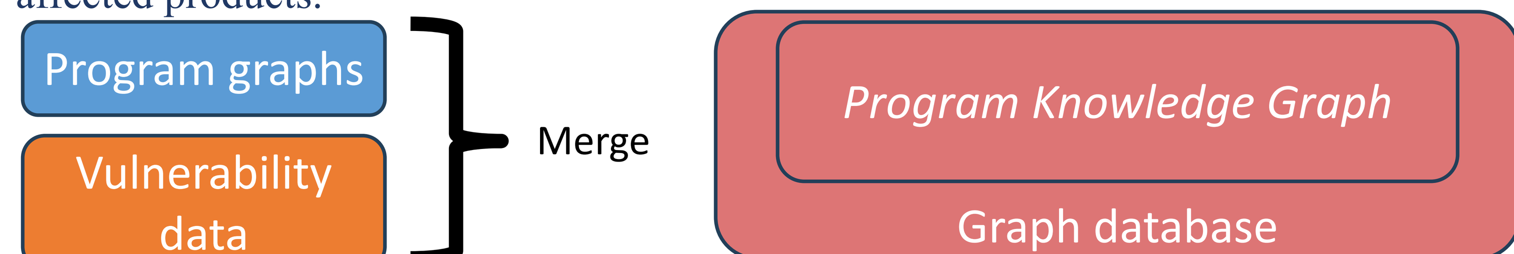
- **Step 1:** Known vulnerability data pertaining to Common Vulnerabilities and Exposures (CVE) and Common Weakness Enumeration (CWE) are methodically sourced from a vulnerability database and stored in a graph database.



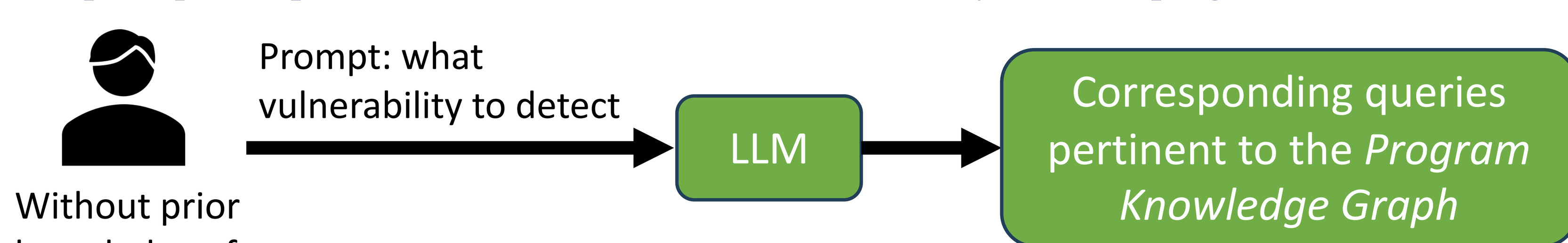
- **Step 2:** Software programs can be comprehensively analyzed and understood using graphs, where nodes represent program components, and edges depict relationships or flows between them.



- **Step 3**: *Program Knowledge Graph* amalgamates vulnerability data and program graph, storing them in a graph database. Each node within this graph represents either a security data entity or a procedure call, with each edge signifying the relationships between source and destination nodes. Nodes and edges can also have attributes to store additional information, such as severity scores and affected products.



- **Step 4:** We utilize Large Language Model (LLM) prompts to generate queries for checking vulnerabilities in the program. Precisely, developers and security analysts can use LLMs to generate complex queries pertinent to the intersection of vulnerability data and programs.



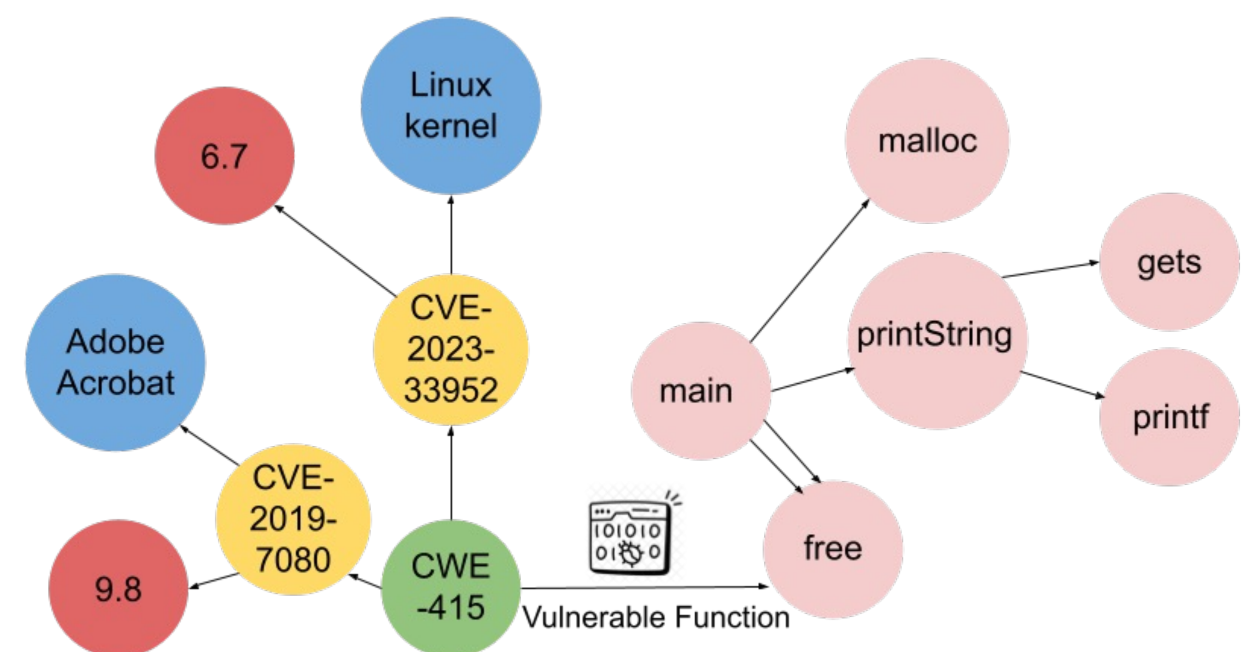


Figure 2: An example of *Program Knowledge Graph* contains the call graph of the example C code in Figure 3 on the right side (colored in pink) and the security data displayed on the left side with CWE-415 pointing to a function call *free* causing *double free* vulnerability.

```
void main() {
    char* ptr = (char*)malloc(8);
    printString(ptr);
    free(ptr);
    free(ptr);
}
void printString(char* ptr) {
    gets(ptr);
    printf(ptr);
}
```

Figure 3: C code example with the vulnerability of calling *free* twice on the same memory address.

```
MATCH (cwe:CWE {`CWE-ID`: "CWE-415"})
MATCH (node:CallGraph {Name: cwe.`events`})
WITH node.A AS arg1, COLLECT(node) AS sarg1
WITH arg1, sarg1, SIZE(sarg1) AS nCount
WHERE nCount > 1
UNWIND sarg1 AS vulNodes
MATCH path=(CallGraph {Name: "main"})-[*]->(vulNodes)
RETURN path
```

Figure 4: Example of a Cypher query to detect a *double free* vulnerability in the code example in Figure 3.

An example of *Program Knowledge Graph* is shown in Figure 2, where nodes colored in pink show the call graph of the code example in Figure 3 while other nodes comprise the extracted vulnerability data. The arrows within the call graph symbolize calling relationships. *Program Knowledge Graph* creates additional edges connecting nodes from the vulnerability dataset to the vulnerable function calls extracted from the program.

Figure 4 showcases a Cypher query that identifies the weakness of CWE-415 *double free* in the C code shown in Figure 3. Specifically, the query checks if a program calls *free* twice on the same memory address, potentially corrupting the program's memory management data structures.

03 EVALUATION

CWE	Name	Program events	Code samples
CWE-242	Use of inherently dangerous function	gets, atoi, atol, atof	4
CWE-401	Missing memory release after effective lifetime	malloc	1
CWE-415	Double free	free	2
CWE-467	Use of sizeof() on a pointer type	sizeof	1
CWE-477	Use of obsolete function	getpw, auto_ptr	3
CWE-479	Signal handler use of a non-reentrant function	syslog	1
CWE-558	Use of getlogin() in multithreaded application	getlogin	1
CWE-1341	Multiple releases of same resource or handle	fclose, free	2

Table 1: Our queries for the *Program Knowledge Graph* successfully identified 14 out of 15 weaknesses in the code samples we collected from 8 categories of CWEs.

To evaluate our proposed method, we collected **15 code examples in C/C++** as benchmarks from CWE sites associated with **8 common software weaknesses**, as shown in Table 1.

The process of our evaluation

- We leverage Neo4j, a popular open-source graph database, as our backend for graph traversal.
- We provide the CSV formatted dataset of CVEs and CWEs, along with the call graphs.
- By utilizing Cypher, we successfully identified the weakness of **14 out of 15 benchmarks**.
- Limitation: We could not identify the weakness of CWE-401, as it requires the inclusion of data flow graphs within the program graph, as call graphs do not suffice. We plan to expand our approach to include various program graph types to detect more complex vulnerabilities and extend our evaluation by comparing them with existing vulnerability detection methods.

04 CONCLUSION

We propose the concept of *Program Knowledge Graph* by integrating program graph and security data and subsequently auto-generating queries by leveraging LLM's prompt tuning to discover vulnerabilities within software code. We are going to continue working on this research project to further investigate the systematic methodology of automating the query generation process by leveraging LLMs.

RESEARCH POSTER PRESENTATION DESIGN © 2019
www.PosterPresentations.com