Causative Insights into Open Source Software Security using Large Language Code Embeddings and Semantic Vulnerability Graph

Nafis Tanveer Islam
Department of Computer Science
University of Texas at San Antonio

Gonzalo De La Torre Parra
Department of Cyber Security Systems
University of the Incarnate Word

Dylan Manuel
Department of Computer Science
University of Texas at San Antonio

Murtuza Jadliwala
Department of Computer Science
University of Texas at San Antonio

Peyman Najafirad
Department of Computer Science
University of Texas at San Antonio

***Abstract*—Open Source Software (OSS) security and resilience are worldwide phenomena hampering economic and technological innovation. OSS vulnerabilities can cause unauthorized access, data breaches, network disruptions, and privacy violations, rendering any benefits worthless. While recent deep-learning techniques have shown great promise in identifying and localizing vulnerabilities in source code, it is unclear how effective these research techniques are from a usability perspective due to a lack of proper methodological analysis. Usually, these methods offload a developer's task of classifying and localizing vulnerable code; still, a reasonable study to measure the actual effectiveness of these systems to the end user has yet to be conducted. To address the challenge of proper developer training from the prior methods, we propose a system to link vulnerabilities to their root cause, thereby intuitively educating the developers to code more securely. Furthermore, we provide a comprehensive usability study to test the effectiveness of our system in fixing vulnerabilities and its capability to assist developers in writing more secure code. We demonstrate the effectiveness of our system by showing its efficacy in helping developers fix source code with vulnerabilities. Our study shows a 24% improvement in code repair capabilities compared to previous methods. We also show that, when trained by our system, on average, approximately 9% of the developers naturally tend to write more secure code with fewer vulnerabilities.**

1. Introduction

In the landscape of modern cyber security, Open-Source Software (OSS) has emerged as a critical component in a wide array of systems, ranging from IoT [1] platforms to essential software supply chains [2]. This prevalence positions OSS as a prime target for cyber adversaries. This is evidenced by high-profile breaches like the compromise of SolarWind's Orion platform [3], which affected approximately 18,000 stakeholders, including major government bodies and critical infrastructure providers. Such events not only highlight the vulnerabilities existing in widely-used software libraries [4], which can lead to extensive service disruptions [4], [5], but also uncover the high impacts of cyber intrusions.

In response to these escalating security challenges, the interdisciplinary Open-Source Software Security Initiative (OS3I) was created, including agencies such as CISA, NSF, DARPA, and NIST. OS3I intends to develop software security guidelines and robust security measures [6], which are increasingly important to consider in addressing cyber threats. Nevertheless, emerging technologies, including large language models such as GPT-4, escalate the complexity of ensuring the security of code generation. On the bright side, these technologies can streamline code development, yet they can also introduce significant security challenges [7], [8], specifically by producing vulnerable code that is later used in production.

Given the dual nature of open-source code repositories or large language code generators, they require to be handled with caution. For novice developers, these resources can be invaluable as they facilitate faster progress and broader participation yet simultaneously introduce potential venues for adversaries to inject vulnerabilities into software [9], [10]. Current statistics indicate that nearly 40% of open-source code resources may not meet stringent security standards [11] for a software application. This is a concerning trend as data from these resources is used to train large language code models. This scenario highlights the crucial role of developer proficiency in code security to mitigate these risks. Therefore, it is essential to recognize the challenges faced by developers, with some narratives framing them as potential ”weakest link in the chain” [12] in software security. Current research emphasizes the need for robust security support and developer training [12], [13]. Surveys conducted by Assal et al. [14] and Hermann et al. [15] echo this sentiment, revealing gaps in developers' understanding of security practices and a lack of access to security experts. Proactively addressing these challenges becomes imperative to enhance the overall security posture of software development.

Recent advancements in automated vulnerability detection techniques [16], [17], [18], especially those based on transformer models [19], [20], have shown promise in identifying potential vulnerabilities in static source code. Transformer methods can learn patterns in source code to identify potential vulnerabilities. Further innovations are brought by graph convolutional networks (GCN) [21], which can help security analysts determine specific vulnerability types. Another promising area is automated vulnerability repair (AVR) [22], [23], [24] systems powered by generative models [25], [26], [27], [28], which can autonomously generate recommendations for fixing vulnerable code. Automated vulnerability detection tools are essential for fending off cyber threats. While these tools are essential in mitigating cyber threats, they face challenges such as the diversity or complex business logic of source code, the evolving nature of cyber threats, the lack of comprehensive datasets, and the need for human expertise.

Novel user-focused research suggests that software developers are in need of diagnostic information when addressing security challenges [12], [13]. While current vulnerability classification and localization methods play a critical role in discerning the specific location of code runtime failure, they cannot determine the root cause of the vulnerability. This problem is further exacerbated within large-scale applications, potentially comprising millions of lines of code, where crash reproduction can be exceptionally challenging due to environmental discrepancies of the system [29], [30], [31]. Although capturing program stack traces or logs [32], [33] might provide insights such as vulnerability-causing inputs or code break locations, the derived results are often incomplete as they do not necessarily identify the root cause of the vulnerability. Consequently, these run-time analyses from stack traces are incredibly lengthy; therefore, these require a long time to be sorted out, diminishing the usability of this method since it requires too much back and forth between code writing and full deployment. The lack of precise diagnostic information necessary to effectively rectify vulnerabilities highlights a pressing need for tools that not only identify but also explain the root causes of vulnerabilities, especially in large-scale applications with extensive code bases.

To address these challenges and provide developers with a more comprehensive understanding of vulnerabilities, we conducted an extensive survey with participants majoring in Computer Science from an R1 Research University. The insights from this survey revealed that existing automated vulnerability detection techniques often fail to provide developers with a clear understanding of a vulnerability's root cause as well as its classification and localization. Bridging this gap, we introduce T5-GCN, a novel approach for root cause diagnostics of software vulnerabilities. T5-GCN combines the power of a large language model (LLM) and a graph convolutional network (GCN) to classify and localize a vulnerability as well as identify the root cause, and provide developers with a static description in the source code. Additionally, T5-GCN integrates an explainability-based approach to facilitate extracting relationships between vulnerable statements and the pivotal contributors to the vulnerability's root cause. We conducted another human-centric analysis to evaluate our approach's efficacy and practical implications. The results demonstrate that T5-GCN can effectively aid developers in rectifying vulnerabilities and enhancing their proficiency in crafting more secure code. Moreover, T5-GCN demonstrated robustness in discerning the root causes of N-day and zero-day vulnerabilities. Overall, T5-GCN presents an innovative approach for root cause analysis of software vulnerabilities.

The contributions delineated in this paper are as follows:

- We present an introductory survey analysis to determine the flaws of the current systems and the shortcommings from the developers on security knowledge. Our primary analysis concludes that the root cause of the vulnerability is needed to help developers solve the vulnerability.
- We propose T5-GCN to find the root cause of a vulnerability using explainable techniques along with its classification and location of the existing vulnerability with a short static description of the vulnerability category. Furthermore, we performed an extensive evaluation with human subject participants to measure the effectiveness of our system compared to previous ones.
- We provide numerical analysis to show the quantitative results of our system. Furthermore, we also demonstrate the generalizability of our proposed approach by identifying N-day and 0-day vulnerabilities with their root cause in various open source applications.

Our data and source code are made publicly available here. [1]

2. Background and Motivation

As the complexity of software continues to increase [34], the significance of rigorous code security analysis in the post-deployment stage becomes paramount. In the absence of such measures, we leave our systems at risk, with potential vulnerabilities wide open to threat actors. These actors can exploit backdoors, which, although not deliberately created, may inadvertently remain within the codebase, posing a serious threat to system integrity and user trust. Proactively averting such vulnerabilities before deployment by analyzing static code testing constitutes a relatively less laborious approach, resulting in substantial savings in time and costs. Nevertheless, recent open-source static application security analysis tools exhibit limitations in generalizability. Furthermore, these tools do not offer actionable recommendations and comprehensive explanations to substantiate their decisions for the developers. Therefore, we deploy a survey to a targeted group of participants to further understand a software developer's current knowledge and needs to analyze a vulnerability and the usability of

[1] https://anonymous.4open.science/r/Threat_Detection_Modeling-BB7B/README.md

current SOTA techniques. The results from the survey will help us decide on the drawbacks of developer knowledge along with the challenges of current SOTA methods.

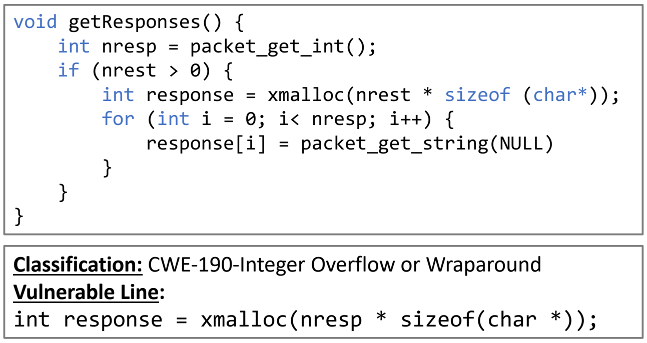
TABLE 1. Demography of the Participants from the Assisted and Controlled Group

	Control	Assisted	Total
Undergraduate			
Junior (Year 1 and 2)	5	6	11
Senior (Year 3 and 4)	6	8	14
Graduate			
MS	6	6	12
PhD	8	11	19
Years of Programming Experience			
0-2 Years	4	7	11
2-5 Years	9	13	22
More than 5 Years	8	9	17
Security Courses Taken			
Yes	1	2	3
No	21	32	53
Total	22	34	← N = 56

Participant Demography. In total, 56 participants completed our survey. We divided these participants into control and assisted groups. The control group only had access to the outputs from the SOTA models, and the assisted group had access to the outcome from our system only. At the end of our survey, we provided the participants with a demographic questionnaire, asking about their educational background, programming experience, and language preference. Table 1 shows the demographic information of our survey participants among two study groups, assisted and control.

The majority of participants are graduate students with CS majors with programming experience of more than two years. However, only one of the 23 participants from the controlled group had taken security-related courses, and two from the assisted group had taken security-related courses. While 5 and 6 participants from assisted and control groups were junior university students, and approximately 81% were senior undergraduate or graduate (MS, PhD) students with more than two years of programming experience. Table 1 presents us that almost 60% of the participant's control and 50% of the assisted participants are graduate students, and approximately 70% had more than two years of programming experience, making them a proper fit for our analysis. Furthermore, these groups represent potential students entering the software industry within a year or two or has some industry experience. Given their background and qualifications, they represent an ideal population capable of gaining proper insights into how the current education system has enriched their code security knowledge while simultaneously providing insights into the benefits and contributions of SOTA methods.

2.1. Investigating Developers' Knowledge in Security



```
void getResponses() {
    int nresp = packet_get_int();
    if (nrest > 0) {
        int response = xmalloc(nrest * sizeof (char*));
        for (int i = 0; i< nresp; i++) {
            response[i] = packet_get_string(NULL)
        }
    }
}
```

Classification: CWE-190-Integer Overflow or Wraparound
Vulnerable Line:
`int response = xmalloc(nresp * sizeof(char *));`

Figure 1. Sample source code provided to the participants depicted at the top and output at the bottom provided by the SOTA methods. We conducted our initial survey by providing the participants with this information and determined their capability to repair vulnerability using these two outputs: classification and vulnerable line.

It is crucial to recognize and address the challenges developers face during code development. They are responsible for making final decisions on application design and implementation procedure, which ultimately reaches production. But a recent research underscores the necessity of providing comprehensive security support and training to empower software developers in effectively mitigating the security issues before code deployment [12], [13]. Another study by Assal et al. [14] delved into the human aspect of source code security, shedding light on the insufficient collaboration between software developers and security experts.

Initially, we aimed to measure the percentage of developers who could write secure and vulnerability-free code without prior support. Therefore, in our initial investigation, we carefully crafted ten skeleton functions for a grocery store shopping cart and provided these to both groups. We also intentionally designed the function definition so that the response by the participants may fall susceptible to one particular type of vulnerability, as shown in Table 2. During this study, we did not provide any information regarding code security or tell them to be cautious about it. Our goal was to observe the natural coding style and quality of the participants and set a baseline to determine whether their natural coding practice included proper security measures. Details of the given task are explained in Section 4.

Table 2 shows the results from the assisted and control groups. From this table, we can see that out of the 56 participants who participated in our initial survey, CWE-

TABLE 2. Success rate of participants on completing ten different functions with proper security measures

Function Name	CWE Number	Success
`calculateCombinations`	119	20%
`extractPrice`	264	40%
`exportPrices`	125	22%
`⋆loadPrices`	200	26%
`printMaxPrice`	416	46%
`validateUserCreation`	399	33%
`addUser`	20	18%
`removeUser`	476	26%
`promptUserCreation`	189	33%
`⋆resizeDatabase`	190	26%

TABLE 3. THE SUCCESS RATE OF PARTICIPANTS FROM THE CONTROL GROUP WHO COULD REPAIR THE FUNCTION ONLY WHEN THE VULNERABLE LINE AND THE CWE CLASS OF THE VULNERABILITY FOR EACH FUNCTION WERE PROVIDED.

Function Name	CWE-Number	Success
getValueFromList	125	47%
*callHelper	416	48%
SQLConnect	264	20%
readFile	416	20%
*createBoard	20	46%

119 and CWE-20 have one of the lowest success rates, 20%, and 18%, respectively. The success rate on the rest of the CWEs is slightly higher and peaks at 46% for CWE 416. The vulnerabilities with the lowest success rate are about writing something out of the memory buffer or performing improper input validation. This relatively poor performance in writing correct and secure code also indicates the developers' indifference to safeguarding some of the variables they are supposed to operate.

2.2. Investigating Usability of Current SOTA Techniques

The usability of current publicly available code security tools is yet to be analyzed systematically. There are various popular static analysis tools such as Infer [35] and Cppcheck [36] or deep learning methods to detect [16] [20], localize [37] [38] [39] and repair [40] [22], which can be used to address the security challenges faced by developers. While these tools and techniques provide some information to classify and localize vulnerability, they are not designed to act as an assistive/interactive tool for humans to understand the vulnerability (embedded in the code) in general or provide solid reasoning for their decision. The ideal goal of such tools should not be to offload a developer's task completely but to act as an assistive tool that would educate developers with appropriate security knowledge to help them become more self-sufficient in writing secure code.

In order to understand how the SOTA methods help developers repair software vulnerabilities, we performed another investigation on the effect of the outcomes produced by current SOTA methods like VELVET [41], VulChecker [39], and LineVul [37] which identify the vulnerability class along with the line of the vulnerability. In this study, we provide the control group of participants with five new vulnerable codes, the vulnerable line in that code, and the CWE classification of the vulnerability, similar to those provided by VELVET, VulChecker, and LineVul. Then, based on the information, we ask the developers to fix the vulnerability in the given code. We aim to determine the percentage of the participants that successfully repaired the vulnerable code. Figure 1 shows a sample code and SOTA outputs provided to the control participants to solve the vulnerability.

For each of the five vulnerable functions, we generate outcomes for each of them from all the current SOTA methods. Our security experts then analyzed the outcomes for each function and determined the correct output. We do this to ensure that none of the outcomes provided to the participants were false positives or false negatives. After ensuring the correctness of all outcomes, we provided the participants with the five functions and the verified outcomes. Table 3 shows that a majority of developers could not fix the given vulnerable code with the provided information. While the classification and localization provided some amount of insight on the code vulnerability to the developers, from the generated outcome by the developers, we can safely assume that these methods helped to solve less than 50% of the vulnerability. While this is an improvement compared to writing code without any suggestions, if we consider a real-world situation, we see a possibility that approximately 50% of code in production is still susceptible to vulnerability.

2.3. Concluding the Findings

While we investigated the challenges of current SOTA techniques to identify the benefits received by the developer, we asked them some follow-up questions to find the gap to assist the developer in writing secure code. We provided the developers with structured questions followed by an open-ended interview to determine what type of system they expected from which they would benefit the most. Since only the control group has seen outcomes from the existing SOTA methods, we asked them some follow-up questions.

In the first question, we wanted to determine the number of participants familiar with CWE vulnerability categories and found that only 17% of participants are familiar. In the following question, we decided to investigate whether they looked for the definitions or the descriptions of the CWE on the internet, and 26% of the participants did not. Furthermore, approximately 68% mentioned that the description would help them greatly instead of looking it online. The last question we asked the participants was open-ended. Here, we asked the developers why they could not solve the vulnerability given the information. Almost 62% of the developers responded that they did not know which line to look for to fix the particular vulnerability. Furthermore, 66% of the developers also mentioned that if they were given the line or the root cause line they have to edit to repair along with the description, it would benefit them heavily.

Our primary analysis demonstrates the lack of developer training on code security. Further, we highlighted the failure of current SOTA techniques in assisting developers in repairing vulnerabilities. From the open-ended interview with the developers, we concluded that if developers are given a system that produces the root cause of the vulnerability with a reasonable description of the vulnerability, it would benefit them greatly.

3. Preliminaries

3.1. Threat Model

In exploring our threat model, we assume that rule or AI-based tools offload the task of vulnerability repair from humans. Therefore, we consider two types of threat actors in the software development landscape. The first category is the human developers who write vulnerable code because of a lack of proper security training on code. The second threat actor category is open-source platforms like ChatGPT, GitHub, or StackOverflow, where the same developers go and reuse code without security concerns.

Our threat model mainly focuses on the vulnerable code developers write in C/C++. Our primary analysis shows that software developers are primarily vulnerable to addressing null pointer or resource management-related vulnerabilities. The vulnerabilities at these levels could compromise the physical memory and CPU caches by gaining privileged root mode access. The attackers could exploit such vulnerabilities through various attack scenarios, like buffer overflow, code injection, improper operations within a memory buffer, or similar vulnerabilities related to these. These vulnerabilities can subsequently provide control over the system, enable data theft, or even launch further attacks. Our approach to minimizing vulnerability in source code is twofold. We propose a system to aid developers in finding the root cause of vulnerability along with the CWE category, complemented by a description of the vulnerability and the vulnerable line. Secondly, with the combination of these pieces of information, we aim to provide usable security to aid developers in writing better and more secure code with the assistance of AI.

3.2. Problem Formulation

In order to assist developers in fixing a vulnerable code and making them aware of introducing future vulnerabilities, in this study, we want to address the problem of finding the root cause of vulnerability along with its CWE classification with a short static description and the vulnerable line. To prepare the input for the GCN, we convert each program function, denoted as p_i, into a multi-edged graph, $\mathcal{G}_{raph}$. This graph is constructed such that the set of nodes, T, represent the programming language tokens, and the edges, E_{dge}, indicate the connections between these nodes. We transform the edge pairs into an adjacency matrix A.

Our methodology employs a multitask function that identifies the root cause of vulnerability with its classification and localization. Furthermore, we provide the developers with a static description curated explicitly for novice developers. Initially, it ascertains the vulnerability category, denoted as CWE, that a vulnerable input function p_i aligns with. After classifying p_i, the model further localizes the starting (L_{start}) and the ending (L_{end}) vulnerable lines, $[L_{start}, L_{end}]$. In the context of our system, we define three outputs: the vulnerability classification CWE with a static description, the vulnerability localization range $[L_{start}, L_{end}]$, and the root cause of vulnerability V_{root}.

Root Cause Analysis through Explainability: The proposed architecture uses explainable artificial intelligence techniques to discern the impacts contributed by each program token, denoted $t_i \in T$, on the vulnerability prediction of a given program.

This model, initially trained to classify and localize the vulnerabilities, garners a holistic understanding of the vulnerabilities and correspondingly attributes weights to each token in the source code. Beyond merely localizing the vulnerable lines, our model proficiently comprehends individual tokens' implications. By prioritizing specific tokens over others, the model inherently puts more significance to the critical tokens that show a higher likelihood of engendering vulnerability.

For the scope of this research, the weight associated with a token t_i is symbolized as ϕ_{i}. The predictive output delivered by our model for one program is mathematically defined as follows:

$$\hat{A} = f(T) = \phi_0 + \sum_{i=1}^{N} \phi_{\mathrm{i}} t_i \tag{1}$$

Where $\hat{A}$ is the set of attributions for each token in T, and N is the total number of tokens. In this expression, the weight ϕ_{i} encapsulates the contribution of a token t_i to the model's overall output.

Assessing System Efficacy: To address the issues we found in our initial analysis in Section 2, we try to address them by answering the following three Research Questions (RQs):

RQ1: *Using the root cause of vulnerability, how effectively is our system assisting software developers in fixing code and educating developers in writing secure code with fewer vulnerabilities?*
To find the root cause of source code vulnerability, we used an explainability-based technique to determine the importance of tokens. Then we propose an in-depth survey analysis to determine how effective our system is compared to the current SOTA methods.

RQ2: *How efficiently can we classify and localize the vulnerability in addition to finding the root cause vulnerability of source codes?*
To find how efficiently we can localize vulnerability, we use a metric called IoU. Furthermore, in order to measure the classification accuracy, we measure F1 Accuracy scores. The higher classification and localization performance will ensure the root cause analysis's effectiveness.

RQ3: *Does our root cause detection system generalize enough to identify zero- and n-day vulnerabilities from the wild?*
To answer this question, we analyzed several open-source projects written in C/C++, and our internal security experts manually checked the validity of the root cause provided by our system.

These RQs serve as a benchmark to evaluate our system's vulnerability detection and localization capabilities, its competency in providing actionable insights for vulnerability remediation, and its adaptability in discerning new vulnerabilities across diverse environments. Subsections 6.4, 6.5 and 6.6 provide in-depth detail in our attempt to answer the above three research questions.

4. User-Focused Survey Design

We designed a user-centric survey to evaluate the effectiveness of our proposed T5-GCN for addressing code vulnerability issues and aiding developers in learning secure coding practices with less reliance on AI tools. Figure 2 shows the entire workflow and architecture of our study and step 1 and 3 in the figure demonstrates the survey analysis presented in this section.

4.1. Participant Recruitment

We recruited our participants from the Computer Science department, enlisting both undergraduate and graduate students who have either completed or are currently enrolled in at least one programming language course. We selected participants with a wide range of educational backgrounds: 1) Undergraduate Students with programming experience, 2) Graduate Students without industry experience, 3) Graduate Students with industry experience, and 4) Graduate Students with research experience.

The recruitment process took place via email invitation and flier distribution, with the approval and support of the department's admin. Furthermore, since many of these participants will join the industry after graduation, their selection is strategic, considering they likely possess limited knowledge and training in code security. The selected group allows us to estimate the effect of insufficient code security knowledge among emerging professionals. Our recruitment process emphasized evaluating their code-writing quality, and therefore, we deliberately omitted the focus on code security practices in our survey. This omission aimed to prevent potential bias, as participants could have otherwise prepared in advance, skewing the results.

Compensation and Approval. Each participant received a compensation of US$25 for their contribution to the study. This study involved human participants and was approved by the institution's Institutional Review Board (IRB).

For a broader understanding of the impact of our proposed system, participants were randomly allocated to two distinct groups: the Control Group and the Assisted Group. Participants were provided with four types of information based on which group they were assigned to: 1) the specific vulnerable line(s) of code, 2) the vulnerability category, 3) the description of the category, and 4) the root cause of the vulnerability. The control group was provided with the vulnerable line and its associated category. In contrast, the assisted group was provided with the root cause of the vulnerability, along with classification, description and localization.

4.2. Survey Design

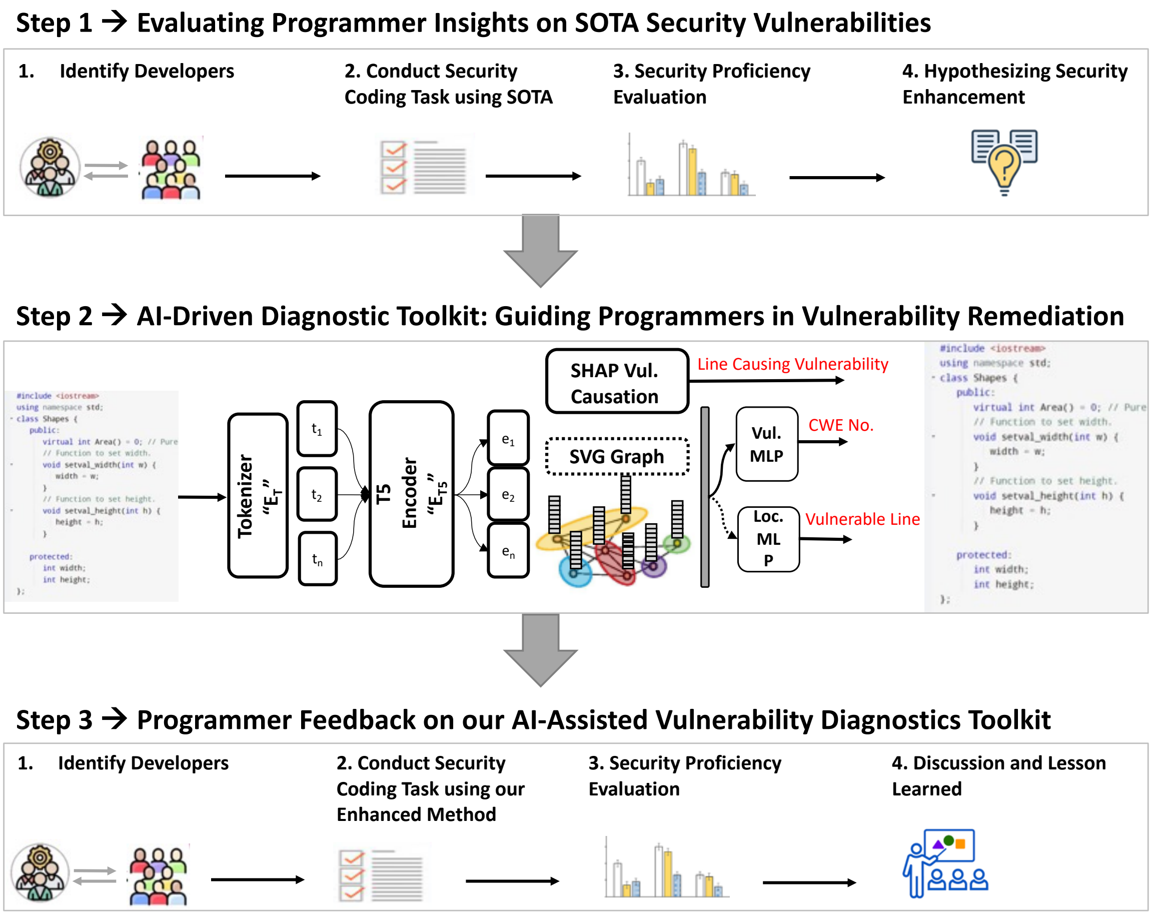
Our survey is segmented into two stages to rigorously evaluate our model's effectiveness and usability, namely 1) The baseline and 2) the evaluation stages.

Baseline Stage. In the baseline stage, we asked participants from assisted and control groups to complete ten C functions associated with a grocery store management system. The participants have to download C source files containing skeletal code (i.e., a function declaration without implementation) for each function, accompanied by implementation instructions, parameter and return value details, and example usages. Participants were required to document the start and completion times for each function. Listing 1 illustrates one of the ten functions the participants need to implement. The tasks involved core C programming concepts like file I/O and memory management. However, we ensured that the implementations of these functions steer clear of advanced data structures to avoid unnecessary complexity. Essential C libraries and user-defined structures were provided with the skeleton code beforehand. Participants were neither encouraged nor discouraged about using and utilizing resources from online or any code-generative AI, and they could use any code editor or IDE based on their personal preferences. The baseline stage is presented in Subsection 2.1

```
long calculateCombinations(long numItems)
{
    /**
     * Problem:
     *  In a grocery store, there are different types of items available for purchase. The store
     * manager wants to calculate the total number of possible combinations of items that a
     * customer can buy. This function should return the factorial of numItems (i.e. numItems!)
     *
     * Example:
     *  calculateCombinations(3) returns 6 (3! = 3 * 2 * 1 = 6)
     *
     * Parameters:
     *  numItems: the total number of items in the store.
     *
     * Returns:
     *  the total number of possible combinations.
     */
    //  --> PARTICIPANT: ENTER HERE TIME STARTED <--
    return 0;
}
//  --> PARTICIPANT: ENTER HERE TIME COMPLETED <--
```

Listing 1. A function that primarily checks if the participant handled integer overflow and properly set up a base case to avoid a stack overflow if solved recursively.

Furthermore, we provided the control group with five functions intentionally associated with a particular CWE number. This means that if the participant makes any security mistake, it will be tied to that CWE number. We provided the two outcomes provided by the SOTA techniques. They were the vulnerable line and the CWE category of the vulnerability. The task was to resolve vulnerability with the given information. We conducted this part of the survey in Subsection 2.2.


Step 1 → Evaluating Programmer Insights on SOTA Security Vulnerabilities
1. Identify Developers
2. Conduct Security Coding Task using SOTA
3. Security Proficiency Evaluation
4. Hypothesizing Security Enhancement
Step 2 → AI-Driven Diagnostic Toolkit: Guiding Programmers in Vulnerability Remediation
Tokenizer "E_T"
t1
t2
tn
T5 Encoder "E_T5"
e1
e2
en
SHAP Vul. Causation
Line Causing Vulnerability
SVG Graph
Vul. MLP
CWE No.
Loc. MLP
Vulnerable Line
Step 3 → Programmer Feedback on our AI-Assisted Vulnerability Diagnostics Toolkit
1. Identify Developers
2. Conduct Security Coding Task using our Enhanced Method
3. Security Proficiency Evaluation
4. Discussion and Lesson Learned

```
#include <iostream>
using namespace std;
class Shapes {
  public:
    virtual int Area() = 0; // Pure
    // Function to set width.
    void setval_width(int w) {
      width = w;
    }
    // Function to set height.
    void setval_height(int h) {
      height = h;
    }

  protected:
    int width;
    int height;
};
```

```
#include <iostream>
using namespace std;
class Shapes {
  public:
    virtual int Area() = 0; // Pure
    // Function to set width.
    void setval_width(int w) {
      width = w;
    }
    // Function to set height.
    void setval_height(int h) {
      height = h;
    }

  protected:
    int width;
    int height;
};
```

Figure 2. Our proposed approach is organized into three pivotal steps (1) Evaluating programmer insights on current state-of-the-art security vulnerabilities; (2) Introducing an LLM-powered diagnostic tool that assists programmers in vulnerability remediation; and (3) Analyzing programmer feedback on the toolkit's effectiveness.

Evaluation Stage. In this stage, we assess the participants' ability to repair code and extend the knowledge they have acquired regarding the rectification of security-related code vulnerabilities. The assessment revolves around the five functions we previously provided to the control group in subsection 2.2. However, this time, the outcomes we provided to the assisted group were the vulnerable line, the CWE class with description, and the root cause of vulnerability. We asked the assisted group to repair the given five codes. Furthermore, we asked the developers from both groups to rewrite the ten functions they wrote in the baseline stage in Subsection 2.1 to fix any possible security issues. The goal was to determine how developers from both groups improved their knowledge using SOTA methods in the control group compared to our technique in the assisted group. This part was done in Subsection 6.4.

We described the workflow of our survey in Appendix A.1.

4.3. Code Assessment

We employed a dual-method evaluation approach to assess the participants' code regarding security and functionality. Furthermore, we categorized identified bugs according to their CWE classification [42].

Run-time Analysis. For the run-time of the code written by the participants, we generated two sets of test cases for each function—the first set aimed at evaluating the functionality, ensuring that the code meets the intended requirements. The second, more intricate set was designed to unveil potential vulnerabilities, with specific inputs crafted to induce run-time errors if vulnerabilities were present. A function passing all test cases, functionality, and security was deemed free of vulnerabilities.

Manual Analysis. While most participants submitted a fully working code, some provided partially written non-compilable code. Furthermore, some participants also use pseudocode as a solution to vulnerability. Therefore, we also do a manual code review by our internal security experts to identify potential vulnerabilities and fixes. Our security experts initially manually checked the functionality by reading code and try to estimate the outcome based on some input test cases. Furthermore, for security testing, our security experts check whether the function would fall into a runtime error in the case of NULL, negative, or overloading numerical limits in the code. Based on the partially written code or pseudocode, if the function is possibly breaking at these inputs, the security experts deem it as a vulnerable function.

5. Proposed System Architecture

We provide an end-to-end system to analyze source code vulnerabilities and demonstrate our system's capability to assist developers in writing secure and vulnerability-free code. Figure 2 depicts the overall architecture of our proposed vulnerability resolution and evaluation procedure. Step 2 in this figure explains the methodology for code vulnerability classification, repair, and finding the root cause of the vulnerability.

5.1. Code Vulnerability Detection and Classification

For code vulnerability detection and classification, the input source code has to go through pre-processing steps and, finally, through our proposed T5-GCN.

Source Code Representation. In this pre-processing step, the input is an entire function of source code, which may be vulnerable or non-vulnerable. We initially employ the CodeT5 [26] tokenizer, which tokenized words using a byte-pair fashion [43]. CodeT5 tokenizer was pretrained in programming languages like C/C++ to extract the set of tokens T from a given function p_i.

We analyzed the individual functions by random sampling from our datasets and found that the average number of tokens is approximately 490. Therefore, we propose to select 512 as the maximum number of tokens, and we trim the length of the set of tokens T to 512. Moreover, we add two unique tokens, $< BOS >$ and $< EOS >$, at the beginning and end of the program as a separator. If the length of the program is less than 512, we use a unique token $< PAD >$ to resize the length to 512. After finalizing the nodes, we develop $\mathcal{G}$ by connecting the nodes using SVG [20].

Source Code Semantic Graph Representation. We have refined the process of root cause analysis within source code by enhancing the capabilities of Graph Convolutional Networks (GCN) through the incorporation of a Semantic Vulnerability Graph (SVG) [20]. The SVG combines four distinct categories of edges, encompassing data [16], control [16], sequential [44], [45], and poacher flow [20] relationships. These four edge categories comprehensively capture the source code's syntactic and semantic attributes.

By combining these diverse graph types, the GCN gains an intricate understanding of the source code, enabling a contextual interpretation and passive runtime understanding [20] of the static code. This contextualization facilitates the creation of relational representations for different program tokens, significantly enhancing the system's ability to pinpoint the tokens responsible for underlying vulnerabilities. This holistic approach to vulnerability analysis represents a significant advancement in source code security analysis.

CodeT5 Encoder. Our proposed system is powered by CodeT5 [26], a large language model that adopts the encoder-decoder architecture inspired by T5 [25]. It effectively captures the syntactic structure of code and utilizes positional information associated with each token to facilitate token-based localization.

We use the encoder of CodeT5, which consists of multiple layers of self-attention and feed-forward neural networks, to generate the embedding of each node or token in our graph. The self-attention mechanism computes attention weights to capture the input sequence's interdependencies and relationships between elements. This allows for encoding contextual information from the nearby tokens in the code. The output from the self-attention layer is then passed through a feed-forward neural network, which applies a nonlinear transformation independently at each position and finally outputs an embedding vector E of size 768 for each token. This embedding vector acts as the node representation for each token, which is then converted into an adjacency matrix using SVG [20] and passed to the GCN layer.

T5-GCN. Graph Convolution Network (GCN) attempts to comprehend the correlation between any pair of node embeddings of tokens from code we got from the CodeT5 encoder. We introduce a two-layered GCN with a residual connection. Mathematically, we implemented GCN as follows:

$$F_{GCN} = H^{(\mathrm{n+1})} = H^{\mathrm{n}} + \sigma\Bigg(W^{n}_{GCN} H^{\mathrm{n}} A\Bigg) \qquad (2)$$

Here, W^{n}_{GCN} represents the learnable weights at the n-th layer, and H^{n} is the feature representation of all tokens T from a function f_i at the n-th layer. $H^{(0)} = E$ and A represents the adjacency matrix. The multiplication of the matrices W^{n}_{GCN}, H_n, and A is followed by an activation function $sigma$ (e.g., $ReLU$). F_{GCN} is the final representation generated by our proposed GCN.

Loss Function. We use the Focal Loss function [46], built on top of CrossEntropyLoss, which can handle possible data imbalance issues as identified by [20] for vulnerability classification purposes. Our Focal Loss function stands thus:

$$FocalLoss(p^t_{rob}) = -\alpha(1 - p^t_{rob})^{\delta} \log(p^t_{rob}) \qquad (3)$$

In this instance, α denotes the balancing factor between the number of vulnerable and non-vulnerable code samples, while p^t_{rob} is the probability distribution of our model's output. We use δ as an adjustable parameter that distinguishes between easy and hard examples [46].

Detection and Classification. For vulnerability detection and classification purposes, we use the feature vector F_{GCN}, produced by our proposed T5-GCN. We added a dense layer after the feature vector layer generated by GCN. The dense layer `Vul. MLP` generates the CWE Number of the vulnerable code if a vulnerability exists, as depicted in Step 2 of Figure 2. If no vulnerability exists, the *Vul. MLP* layer generates an output of 0. Furthermore, for each identified vulnerability classified by a CWE number, we provide a static description of the identified vulnerability.

5.2. Identification of Vulnerable Lines

In order to find the vulnerable lines, our proposed model identifies a block of code by generating the starting and ending lines of the vulnerable code. The second dense layer `Loc. MLP` generates L_{start} and L_{end}, the line range where the vulnerability exists. Therefore, we connect F_{GCN} with another dense layer *Loc. MLP* for finding the vulnerable line.

Since line numbers vary depending on the position of the vulnerable line in code, we designed the identification of vulnerable lines as a regression problem. Hence, we apply Mean Squared Error (MSE) loss for vulnerability localization. Our MSE loss function is defined as follows:

$$MSE = \frac{1}{n}\sum(L - \hat{L})^2 \tag{4}$$

where L is the original outcome and $\hat{L}$ is the outcome from the model.

5.3. Root Cause of Vulnerability

After the model is sufficiently trained to classify and localize the vulnerability, we find the root cause of the vulnerability using our trained model. We employed DeepLift-SHAP attribution scores. We hypothesize that, since the model can effectively classify and localize the vulnerability, we determine to utilize the model's understanding of vulnerability to determine the contribution of each token to find the root cause of vulnerability.

DeefLiftSHAP is an explainability technique for neural networks based on executing a SHAPly [47] variant of the original DeepLift [48]. Combining DeepLIFT and SHAPly, DeepLiftSHAP operates on deep learning frameworks to explain neural network models. We generate attribution scores based on the DeepLiftShap [47], where we generate the code token attribution scores based on our proposed Algorithm 1. We sum up the attribution scores of each token in a line to generate an attribution score for each line. Here, $\hat{A}$ is the set of attribution scores for all tokens in T of a function p_i, where, $\hat{A} \in \{a_1, a_2, ..., a_m\}$. After generating scores for each line or statement, we consider the line with the highest attribution values before V_{Start} as the root cause of the vulnerability.

6. Experiments and Discussions

We aim to provide an LLM-powered code assistant to help developers write more secure code and potentially educate developers in writing vulnerability-free code. Therefore, in our experiments, we initially measure our system's qualitative and quantitative results, followed by an analysis of its generalizability.

Algorithm 1 Token Attribution for Root Cause Vulnerability

```
Input: $model$, input program $p_i$
Output: Explainable attribution scores $\hat{A}$ for tokens $T$
$T$ = $Tokenizer(p_i)$
$\hat{A}$ = [$t_i$: 0 for $t_i$ in $T$]
$original_{pred}$ = $model(T)$
$contrib_{subset}$ = []
for each $t_i$ in T do
    $contribution$ = DeepLIFT($t_i$)
    $contrib_{subset}.append(contributions)$
end for
for Each $s_{sub}$ in $contrib_{subset}$ do
    $subset_{pred}$ = $model(s_{sub})$
    $marginal_{contr}$ = $original_{pred}$ - $subset_{pred}$
    $norm_{contr}$ = $marginal_{contr}$ / $len(s_{sub})$
    $\hat{A}.append(norm_{contr})$
end for
return $\hat{A}$
```

6.1. Experimental Datasets

D2A and BigVul. Since we provide an end-to-end solution for vulnerability analysis, we only focus on the datasets containing source code from real-world applications. We use Big-Vul [49] and D2A [50] for vulnerability classification, localization, and root cause analysis. BigVul provides ten vulnerability categories that fall within the top 25 CWE vulnerabilities mentioned at CWE [42]. D2A contains open-source projects from GitHub, and labels were created using commit filtering and static analyzer tools. The BigVul dataset provides CWE numbers of vulnerabilities, while D2A does not provide a vulnerability class, so we use it only for vulnerability detection and localization.

IoT OS Repositories. We collected a dataset from six OS repositories from GitHub to test the capability of our system in a real-world operational analysis for wild N-day and zero-day program samples. First, we downloaded six IoT operating repositories: TinyOS, Contiki, Zephyr, FreeRTOS, RIOT-OS, and Raspberry Pi OS. Next, we scanned the entire repository of these operating systems using JOERN [51], a tool specially designed to monitor and analyze large repositories. Then, we used JOERN command line interface to split the C/C++ files into functions for operational analysis.

6.2. Evaluation Metrics

We analyze the qualitative result of our root cause analysis from our survey. However, for quantitative analysis, we use some standard metrics. We achieve vulnerability localization by establishing a boundary between vulnerable lines (starting and ending vulnerable lines). Therefore, we employ the Intersection of Union (IoU) as our evaluation metric. Since our input data consists of source code, which is linear single-dimensional data, unlike a 2D image, we modify the IoU formula [52] to 1D scale. Let's consider our model predicts the localization line boundaries from $V\hat{ul}_{Code} = [L_{\hat{Start}} - L_{\hat{End}}]$ and $Start <= End$ and the ground truth for localization is $Vul_{Code} = (L_{Start}, L_{End})$. So the IoU is:

$$IoU = \frac{|V\hat{ul}_{Code} \cap Vul_{Code}|}{|V\hat{ul}_{Code} \cup Vul_{Code}|} \tag{5}$$

If the value of IoU is zero, Equation 5 demonstrates that $\hat{L}$ and L do not overlap, indicating that the model cannot accurately localize a single vulnerable line. If the value is 1, the model can accurately anticipate all vulnerable lines. However, we used the standard metrics including accuracy, precision, recall, and F1 score metrics for vulnerability classification purposes.

6.3. Experiments

We randomly divided our datasets into 80:10:10 ratios for training, validation, and testing during the experiment. We utilized a 12-layer CodeT5 encoder to generate the embeddings and a two-layer GCN with a residual connection to generate feature vectors for each function p_i. The final feature representation vector produced by GCN is of size 512. We trained the model for 20 epochs with a maximum token length of 512 for each function using a learning rate 6e-6. We used eight A100 NVIDIA GPUs for our experiments.

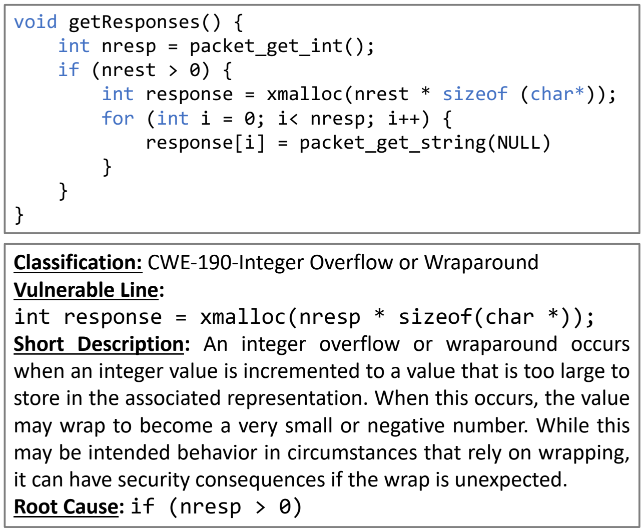
Multitask Training. As depicted in Figure 2 during the training stage, we perform classification and localization for vulnerability analysis. We use cross-entropy loss for vulnerability detection and MSE loss for localization purposes. Furthermore, we provide a static description of the class of vulnerability to the developers.

Since we are identifying ten vulnerability classes, there are ten layers of neurons for vulnerability classification. However, we also provide localizing vulnerability by providing the line range of the statements where the vulnerability exists. Therefore, we put two neurons for vulnerability localization to generate the line numbers for the first and last lines of the vulnerable statements. Finally, for root cause analysis, we used the trained model and utilized explainability to the root cause of the vulnerability.

In the remaining part of this section, we will conduct the experiments by answering the three research questions we defined in Section 3.

6.4. RQ1: Developer Usability Evaluation

To qualitatively evaluate the usability of our system, we used the assisted group of participants to determine how our system can assist them in repairing code vulnerability. Furthermore, we also test the capability of our model to educate the developers in writing vulnerability-free code without assistance. To find the root cause of vulnerability, we use attribution scores generated using SHAPly. The top box in Figure 3 shows the vulnerable code and the bottom box shows the root cause and all the other outcomes generated by our model and provided to the assisted group.



```
void getResponses() {
    int nresp = packet_get_int();
    if (nrest > 0) {
        int response = xmalloc(nrest * sizeof (char*));
        for (int i = 0; i< nresp; i++) {
            response[i] = packet_get_string(NULL)
        }
    }
}
```

Classification: CWE-190-Integer Overflow or Wraparound
Vulnerable Line:
`int response = xmalloc(nresp * sizeof(char *));`
Short Description: An integer overflow or wraparound occurs when an integer value is incremented to a value that is too large to store in the associated representation. When this occurs, the value may wrap to become a very small or negative number. While this may be intended behavior in circumstances that rely on wrapping, it can have security consequences if the wrap is unexpected.
Root Cause: `if (nresp > 0)`

Figure 3. Input code is depicted at the top and output produced our proposed system is depicted at the bottom. Input is the vulnerable code, and our system provides four types of output: 1) Classification, 2) Vulnerable Line, 3) Short Description, and 4) Root Cause.

Assisted Vulnerability Repair. Here, our goal is to determine how our proposed system helps developers fix vulnerabilities in source code. They were given the same five functions the controlled group was given in Subsection 2.2. Our goal is to compare the performance between the controlled and assisted in repairing the vulnerability given the outcome from SOTA methods and the outcome from our system. We provided the participants with the outcome from our model based on the given vulnerable code. Our model's outcome includes the vulnerability classification with a short description, the vulnerable lines, and the root cause of the vulnerability. Figure 3 shows a sample we provided to the participants. We asked the participants to repair the code they were given with the assistance of the outcome provided by our model.

From our analysis in Table 4, we found that 13% of the

TABLE 4. THE SUCCESS RATE IMPROVEMENT OF THE ASSISTED PARTICIPANTS COMPARED TO THE CONTROLLED WHO WERE ONLY GIVEN THE VULNERABLE LINE AND THE CWE CLASS OF THE VULNERABILITY. HERE THE IMPROVEMENT DEPICTS THE PERCENTAGE OF IMPROVEMENT OF THE ASSISTED GROUP COMPARED TO THE CONTROL GROUP FOR VULNERABILITY REPAIR AND HOW QUICKLY THE ASSISTED GROUP REPAIRED VULNERABILITY.

Function Name	CWE Number	Repair Impr.	Time Impr.
`getValueFromList`	125	7%	10%
`*callHelper`	416	15%	6%
`SQLConnect`	264	11%	14%
`readFile`	416	16%	9%
`*createBoard`	20	24%	22%

participants use ChatGPT extensively for code repair. While we assumed this would be a plausible event, we still considered the outcome generated by ChatGPT in our results. By manually analyzing the written code from these 13% of the participants, we found that 18% of the code generated by ChatGPT has vulnerabilities. From this table, we can see that for each CWE, the percentage of assisted participants repairing the code compared to the control group is significantly higher for all cases. The ”Repair Improvement” column from Table 4 shows the improvement percentage of the participants who could repair the vulnerable code compared to the control group.

Furthermore, we also did a time analysis as a part of our survey. In the given code, for each function the participants have written, we asked them to mention the starting and ending time it took them to write the code. Table 4 also shows that, there is a good improvement in the percentage of the assisted participants who solved the vulnerability more quickly. For example, for CWE-20, CWE-264, and CWE-125, the participants took at least 10% less time than their controlled counterparts. This improvement in time shows that our model can significantly boost a developer's time to repair a vulnerability by providing a root cause of the vulnerability and a static description of the CWE.

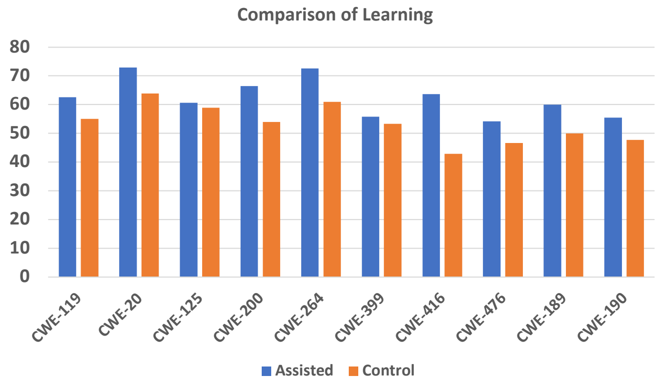


Figure 4. Performance of Developer Education when Comparing Assisted vs. Control Group

Developer Education. In this part, we use the control and assisted group participants to determine how our system eventually educates developers to write vulnerability-free code. Furthermore, the control group has never seen an outcome from our model. Therefore, we hypothesized that by seeing the outcome from our model, the assisted groups have developed the ability to write more vulnerability-free code. In other words, we want to determine how self-sufficient the developers have become in fixing code vulnerabilities entirely.

To test this, we asked both groups of developers with the same ten functions we asked them to reuse the code they completed in Subsection 2.1. Then, we asked both groups to check their code, and based on their learning from the previous phase, they could fix any existing vulnerabilities in their code without any assistance.

Our comparison from Figure 4 shows that the participants from the assisted group have a higher success rate in finding and repairing vulnerabilities from their original code across all CWE types. Furthermore, in the controlled group, there were still vulnerabilities in 49% of the codes they submitted after resubmission. However, in the assisted group, we see only 38% of the code has a vulnerability. Hence, we see an almost 9% improvement in developer learning from assisted groups when using our system compared to SOTA methods. Furthermore, after the survey, 42.3% of the assisted participants responded that they feel more comfortable with analyzing and repairing source code vulnerability, compared to 34% from the control group.

6.5. RQ2: Quantitative Analysis

To quantitatively demonstrate the effectiveness of our vulnerability classifier and find the vulnerable line, we used our proposed T5-GCN model. The task is to classify with an added static description initially and then localize the vulnerable line.

Vulnerability Classification. Initially, we classified vulnerability using our proposed T5-GCN. For the D2A dataset, we compared our model with Devign [41], VELVET [41], and LineVul [37] and PFGCN [20]. For the Big-Vul dataset, we compared our model with VulChecker [39], VELVET [41], LineVul [37], and IVDetect [53]. For the D2A dataset, our model achieves almost a 3% increase in accuracy; however, 10% in F1 score and 15% in recall. For the BigVul dataset, we used the results shared by [37]. Numerical comparison with previous models for the BigVul dataset shows that our model significantly improves F1 and recall. Table 5 shows a detailed comparison of the classification of our T5-GCN model with other models for the two datasets. Since the D2A dataset has no classification information, accuracy, F1, precision, and recall are based on the detection task.

To assess the performance of vulnerability classification using our model, we tasked it to classify 10 categories of vulnerabilities as demonstrated in Figure 5, which frequently appear in the top 25 vulnerable categories in CWE [42], as mentioned by [49].

TABLE 5. VULNERABILITY CLASSIFICATION AND LOCALIZATION WITH BIG-VUL AND D2A DATASET

Data	Model	IoU	Acc.	F1	Pre.	Rec.
D2A	Devign	0.58	-	-	-	-
	VELVET	0.55	0.59	0.58	**0.70**	0.50
	LineVul	0.42	-	-	-	-
	PFGCN	-	0.61	0.61	0.61	0.62
	Ours	**0.72**	**0.62**	**0.68**	0.60	**0.65**
BigVul	VulChecker	0.44	-	0.26	0.18	0.52
	VELVET	0.45	-	-	-	-
	LineVul	0.45	-	0.56	**0.66**	0.60
	IVDetect	-	-	0.35	0.23	0.72
	Ours	**0.49**	**0.65**	**0.62**	0.62	**0.66**

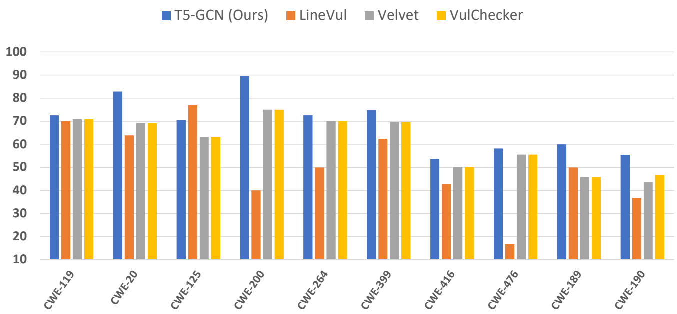


Figure 5. Multi class Vulnerability Classification in Comparison of SOTA Methods to our proposed T5-GCN. X-axis is the vulnerability category, and Y-axis demonstrates the F1 Score

Figure 5 compares the classification F1 score for each CWE category. Furthermore, from our training dataset, we found that CWE-119, CWE-20, and CWE-200 comparatively have the highest number of training samples, having more than 1000 examples. Consequently, CWE-416, CWE-476, and CWE-190 have the least training examples, with less than 400 training samples. Therefore, we see that CWE-119 and CWE-20 achieve an F1 score of 70% and 63.83%, respectively. The classification capability of our model highly outnumbers other SOTA models like VulChecker [39], VELVET [41], and LineVul [37] and PFGCN [20]. However, for other classes like CWE-416, CWE-476, and CWE-190, CWE-190 we see a moderate downfall in performance due to small but still achieving higher performance than SOTA models due to the use of Focal Loss during the training phase of the model.

Finding Vulnerable Line. Based on the prediction by our classifier, we pass the identified vulnerable functions to our vulnerability localizer. Table 5 shows that our localizer produced an IoU of 0.72 and 0.49 IoU, respectively, for D2A and BigVul datasets. Our model produced at least 14% higher IoUs compared to VulChecker [39] on the D2A dataset and at least 4% higher on the BigVul Dataset. We compared our model with other models like VELVET [41] and LineVul [37]. We observe that our model with SVG as an input graph produced a significantly higher IoU score in all the cases.

TABLE 6. Generalizability Testing of our Model. We test with out-of-sample data during evaluation to test the generalizability of the model

Training Data	**Evaluation Data**	**IoU**	**Acc.**	**F1**	**Pre.**	**Rec.**
D2A	D2A	0.72	0.62	0.68	0.68	0.65
	BigVul	0.40	0.52	0.55	0.57	0.52
BigVul	BigVul	0.49	0.65	0.62	0.66	0.66
	D2A	0.44	0.57	0.59	0.60	0.51
Combined	D2A	0.74	0.66	0.75	0.70	0.70
	BigVul	0.50	0.64	0.66	0.67	0.70
	Combined	0.65	0.68	0.64	0.66	0.69

TABLE 7. N-day Vulnerability Discovery and Localization from IoT OS Repositories

IoT OS	N-Day Vulnerability	Count
TinyOS	N/A	0
Contiki	CWE-119	4
	CWE-189	1
Zephyr	CWE-264	2
	CWE-119	4
	CWE-399	1
FreeRTOS	CWE-119	1
	CWE-190	2
RIOT-OS	CWE-119	5
	CWE-476	1
Raspberry-Pi OS	CWE-200	2
	CWE-119	1

6.6. RQ3: Generalizability Testing

Generalizability Testing: We test our model's generalizability by testing the model's outcome on IoT device operating systems. To achieve this, we trained our model with one dataset and tested the model's performance on a different dataset to observe our model's generalization efforts. Furthermore, we combined the D2A and BigVul datasets to achieve more generalizability and tested the model's performance. When we trained the model on the D2A dataset and tested it on the BigVul dataset and vice versa, we can see from Table 6 that the accuracy has decreased. However, when we trained the model with the combined dataset, we saw that the F1 scores had increased up to 7% and the IoU score had increased by 2% for D2A dataset. Moreover, the IoU score has increased 1% for BigVul Dataset. We observe that when we use the combined dataset for training, metrics for D2A increase significantly since D2A dataset is 26 times smaller than BigVul, hence providing more generalizability to our model.

N-day and Zero-day Analysis: We implement a vulnerability localization workflow on six C/C++ based IoT operating system repositories from GitHub to discover wild N-day vulnerabilities. Initially, we scan the repositories using JOERN to extract the functions. Table 7 shows that from the six repositories, we could detect 24 vulnerabilities; however, we could not detect any N-day vulnerabilities from TinyOS. Moreover, from Table 7, we can see that CWE200 and CWE-120 are spread along multiple repositories, providing us with an insight into what kind of vulnerabilities are prevalent across IoT devices.

We found three zero-day vulnerable code samples from the IoT device operating systems. Our model was able to classify and find the vulnerable line of these three zero-day samples. In this case, our model was also trained on the combined dataset consisting of D2A and BigVul. We verified the outcome of our model with security experts. Out of these three confirmed zero-day vulnerabilities, two of them were from Zephyr, and one was from FreeRTOS. All of the vulnerabilities we categorized as CWE-119.

7. Related Work

A recent study done by Werkme [54] suggests that addressing vulnerabilities constitutes a noteworthy issue in open-source software (OSS) because many maintainers ignore minor security vulnerabilities and only address significant concerns as they are brought up by the community or by automated security software. One such automated security software, V1SCAN, was developed by Woo [55] to scan for OSS reuse, a common vulnerability introduced in OSS when maintainers copy out-of-date vulnerable OSS code.

In this section we focus on identifying areas where current research falls short in analyzing vulnerabilities in OSS and setting the stage for our new approach. Current detection tools such as Devign [16], VELVET [41], and LineVul [37] are helpful in identifying source code Vulnerabilities. However, they don't offer enough feedback to guide developers in fully addressing these vulnerabilities.

Advancements in Vulnerability Detection. Vulnerability detection has progressed from conventional machine learning (ML) techniques to more flexible and universally applicable deep learning-based solutions. As Lin et al. [56] elucidates, ML methods offer a promising avenue for automated vulnerability discovery. Simultaneously, deep learning-based solutions such as VulDeepecker [17] and μVulDeepecker [57] offer universal applicability; nevertheless, they presented a heavy reliance on feature engineering for vulnerability detection. The advancements of Devign [16] contrasted these approaches compared to VulBERTa [58], and ReVEAL [59], which used Code Property Graph (CPG) [60] based techniques to identify vulnerabilities, offering improved flexibility. Alongside these, Abstract Syntax Tree (AST) based techniques were proposed by Bilgin et al. [61] and others [62], [63], [64], to retain the syntactic information of the source code during detection. Furthermore, transformer-based models like RoBERTa [65] were adopted by VulBERTa [58] and [66] to detect vulnerabilities in source code. Lastly, Islam et al. [20] proposed a semantic understanding of programming languages to classify and detect vulnerabilities.

Methods of Source Code Vulnerability Localization. The task of localizing vulnerabilities within source code has seen the application of various methods, from traditional rule-based static analysis tools to innovative deep learning techniques. While tools like Cppcheck [36], FlawFinder [67], RATS [68], and Infer [35] provide direct approaches to vulnerability localization, their high false positive and false negative rates [69] underscore the need for more reliable methods. A promising avenue is provided by the use of deep learning, as evidenced by the fine-grained vulnerability detection and locator systems proposed by Vuldeelocator [18] and DeepLineDP [38], leveraging bidirectional Recurrent Neural Networks (RNN). Additionally, the ensemble graph-transformer learning approach by VELVET [41] and the non-conventional explainability technique by LineVul [37] demonstrate advances in detecting and localizing vulnerabilities at the statement level. VulChecker [39] presents a unique approach, using an intermediate representation called LLVM for vulnerability localization.

Challenges and Evolution of Root Cause Analysis. Identifying the root cause of vulnerabilities within a system is a formidable task, especially given the disconnect between states where code crashes and actual root causes. Over time, several notable approaches have emerged. AutoPaG [70], for instance, leveraged the data flow of programs to analyze vulnerabilities such as out-of-bound, buffer overflow errors, and general boundary condition errors. To isolate the root cause of thread scheduling issues, Choi et al. [71] proposed narrowing down differences in thread schedules, resulting in the failure of the program. Further, Failure Sketching [72] introduced a cooperative adaptation of static and dynamic analysis for identifying root causes of production failures. More recent contributions, such as ARCUS [73] and Yagemann et al.'s methodology [74], automate root cause analysis through execution flags and binary level analysis, and tracking execution traces, respectively, enhancing the detection of issues like buffer overflow and use-after-free exploits.

Explainability for Development-Phase Vulnerability Analysis. While the aforementioned methods focus on vulnerability analysis during a program's execution, it is equally critical, from a developer's perspective, to scrutinize vulnerabilities during the development phase [20] [75]. This has led to the proposal of explainable techniques that pinpoint relevant features contributing to a program's vulnerability [76], [77], [78]. Asm2Seq [79] and VulANalyzeR [80] took this concept further by introducing explainable deep learning-based approaches for identifying binary vulnerabilities in source code. Notably, VulANalyzeR [80] employed an attention-based explainable mechanism to unearth the root cause of vulnerabilities.

To conclude, we have explored various methods for vulnerability analysis, highlighting the shift towards development-phase vulnerability analysis and the use of explainable techniques. However, none of the current methods tries to identify the root cause of the vulnerability while ensuring the usability of their proposed system. Therefore, our research aims to fill the existing gap in finding the root cause of the vulnerability of code with a proper, informative, and usable system.

8. Conclusion

This study presents a comprehensive end-to-end solution to address the lack of a usable vulnerability analysis system by finding its root cause using an explainable technique. Initially, we conduct survey analyses to demonstrate the challenges a developer faces from the SOTA methods. We further interviewed the developers to identify the gaps that must be addressed to mitigate the current challenges. We also present classification, localization, and a short description to aid the developer in repairing vulnerabilities. Our approach leverages the combined power of LLM CodeT5 and a GCN in a multitask setting to analyze source code. Our proposed model accurately classifies vulnerability categories, localizes vulnerable lines, and identifies underlying causes across multiple datasets. Furthermore, we test our system's capability by comparing our outcome's effectiveness vs. the outcome from the SOTA methods. Our model exhibits generalizability by detecting N-day and zero-day vulnerabilities in IoT operating system source code. Although, in our survey, we tried to the best of our ability to provide a realistic scenario for code development, this needs to reflect the development condition in a real-life situation holistically. Development in real-life conditions constitutes a complex environment with multiple developers writing code in multiple files, which further increases the chances of vulnerability. In our future work, we aim to extend our methodology by extending the root cause capabilities of our model and combining it with generative models to suggest a possible fix for the developers.

References

[1] Abdullah Al-Boghdady, Mohammad El-Ramly, and Khaled Wassif. idetect for vulnerability detection in internet of things operating systems using machine learning. *Scientific Reports*, 12(1):1–12, 2022.

[2] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. Sok: Taxonomy of attacks on open-source software supply chains. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1509–1526. IEEE, 2023.

[3] Sean Peisert, Bruce Schneier, Hamed Okhravi, Fabio Massacci, Terry Benzel, Carl Landwehr, Mohammad Mannan, Jelena Mirkovic, Atul Prakash, and James Bret Michael. Perspectives on the solarwinds incident. *IEEE Security & Privacy*, 19(2):7–13, 2021.

[4] Log4j, https://nvd.nist.gov/vuln/detail/CVE-2021-44228.

[5] Synopsys. Open source security and risk analysis report. 2023.

[6] US Government. Federal register. 2023.

[7] Raphaël Khoury, Anderson R Avila, Jacob Brunelle, and Baba Mamadou Camara. How secure is code generated by chatgpt? *arXiv preprint arXiv:2304.09655*, 2023.

[8] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. Security implications of large language model code assistants: A user study. *arXiv preprint arXiv:2208.09727*, 2022.

[9] Jukka Niiranen. Democratizing code, https://jukkaniiranen.com/2021/04/democratizing-code/.

[10] AKILEK Akilek Consulting. Democratizing programming: How ai enables everyone to become a programmer, https://www.linkedin.com/pulse/democratizing-programming-how-ai-enables-everyone-become/.

[11] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE, 2022.

[12] Matthew Green and Matthew Smith. Developers are not the enemy!: The need for usable security apis. *IEEE Security & Privacy*, 14(5):40–46, 2016.

[13] Yasemin Acar, Sascha Fahl, and Michelle L Mazurek. You are not your developer, either: A research agenda for usable security and privacy research beyond end users. *2016 IEEE Cybersecurity Development (SecDev)*, pages 3–8, 2016.

[14] Hala Assal and Sonia Chiasson. 'think secure from the beginning' a survey with software developers. In *Proceedings of the 2019 CHI conference on human factors in computing systems*, pages 1–13, 2019.

[15] Charles Weir, Ben Hermann, and Sascha Fahl. From needs to actions to secure apps? the effect of requirements and developer practices on app security. In *29th USENIX security symposium (USENIX security 20)*, pages 289–305, 2020.

[16] 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.

[17] 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. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[18] Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. Vuldeelocator: a deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing*, 2021.

[19] Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Trung Le, Quan Hung Tran, and Dinh Phung. ReGVD: Revisiting graph neural networks for vulnerability detection. In *Deep Learning for Code Workshop*, 2022.

[20] N. Islam, G. Parra, D. Manuel, E. Bou-Harb, and P. Najafirad. An unbiased transformer source code learning with semantic vulnerability graph. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, pages 144–159, Los Alamitos, CA, USA, jul 2023. IEEE Computer Society.

[21] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. *CoRR*, abs/2009.08366, 2020.

[22] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356. IEEE, 2023.

[23] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 5131–5140, 2023.

[24] Zimin Chen, Steve Kommrusch, and Martin Monperrus. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering*, 49(1):147–165, 2022.

[25] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[26] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[27] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[28] Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (stop): Recursively self-improving code generation. *arXiv preprint arXiv:2310.02304*, 2023.

[29] Mihir Bellare and Bennet Yee. Forward integrity for secure audit logs. Technical report, Citeseer, 1997.

[30] Frank Capobianco, Rahul George, Kaiming Huang, Trent Jaeger, Srikanth Krishnamurthy, Zhiyun Qian, Mathias Payer, and Paul Yu. Employing attack graphs for intrusion detection. In *Proceedings of the New Security Paradigms Workshop*, pages 16–30, 2019.

[31] Xueyuan Han, Thomas Pasquier, Adam Bates, James Mickens, and Margo Seltzer. Unicorn: Runtime provenance-based detector for advanced persistent threats. *arXiv preprint arXiv:2001.01525*, 2020.

[32] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 377–390, 2017.

[33] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. Dta++: dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[34] Mamdouh Alenezi and Mohammad Zarour. On the relationship between software complexity and security. *arXiv preprint arXiv:2002.07135*, 2020.

[35] Infer. Infer. 2013.

[36] Cppcheck. https://cppcheck.sourceforge.io/. 2022.

[37] Michael Fu and Chakkrit Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. 03 2022.

[38] Chanathip Pornprasit and Chakkrit Tantithamthavorn. Deeplinedp: Towards a deep learning approach for line-level defect prediction. *IEEE Transactions on Software Engineering*, 2022.

[39] Yisroel Mirsky, George Macon, Michael Brown, Carter Yagemann, Matthew Pruett, Evan Downing, Sukarno Mertoguno, and Wenke Lee. Vulchecker: Graph-based vulnerability localization in source code.

[40] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. Vulrepair: a t5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 935–947, 2022.

[41] Yangruibo Ding, Sahil Suneja, Yunhui Zheng, Jim Laredo, Alessandro Morari, Gail Kaiser, and Baishakhi Ray. Velvet: a novel ensemble learning approach to automatically locate vulnerable statements. *arXiv preprint arXiv:2112.10893*, 2021.

[42] CWE. Common weakness enumeration. 2022.

[43] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics.

[44] Lianzhe Huang, Dehong Ma, Sujian Li, Xiaodong Zhang, and Houfeng Wang. Text level graph neural network for text classification. *arXiv preprint arXiv:1910.02356*, 2019.

[45] Yufeng Zhang, Xueli Yu, Zeyu Cui, Shu Wu, Zhongzhen Wen, and Liang Wang. Every document owns its structure: Inductive text classification via graph neural networks. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 334–339, 2020.

[46] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017.

[47] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *Advances in neural information processing systems*, 30, 2017.

[48] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. Learning important features through propagating activation differences. In *International conference on machine learning*, pages 3145–3153. PMLR, 2017.

[49] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. Ac/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 508–512, 2020.

[50] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. D2a: a dataset built for ai-based vulnerability detection methods using differential analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 111–120. IEEE, 2021.

[51] Joern: The bug hunters workbench. https://joern.io/.

[52] Hamid Rezatofighi, Nathan Tsoi, JunYoung Gwak, Amir Sadeghian, Ian Reid, and Silvio Savarese. Generalized intersection over union: A metric and a loss for bounding box regression. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 658–666, 2019.

[53] Yi Li, Shaohua Wang, and Tien N Nguyen. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 292–303, 2021.

[54] Dominik Wermke, Noah Wöhler, Jan H Klemmer, Marcel Fourné, Yasemin Acar, and Sascha Fahl. Committed to trust: A qualitative study on security & trust in open source software projects. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1880–1896. IEEE, 2022.

[55] Seunghoon Woo, Eunjin Choi, Heejo Lee, and Hakjoo Oh. {V1SCAN}: Discovering 1-day vulnerabilities in reused {C/C++} open-source software components using code classification techniques. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6541–6556, 2023.

[56] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. Software vulnerability detection using deep neural networks: a survey. *Proceedings of the IEEE*, 108(10):1825–1848, 2020.

[57] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. μ vuldeepecker: A deep learning-based system for multiclass vulnerability detection. *IEEE Transactions on Dependable and Secure Computing*, 18(5):2224–2236, 2019.

[58] Hazim Hanif and Sergio Maffeis. Vulberta: Simplified source code pre-training for vulnerability detection. *CoRR*, abs/2205.12424, 2022.

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

[60] 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*, pages 590–604. IEEE, 2014.

[61] Zeki Bilgin, Mehmet Akif Ersoy, Elif Ustundag Soykan, Emrah Tomur, Pinar Çomak, and Leyli Karaçay. Vulnerability prediction from source code using machine learning. *IEEE Access*, 8:150672–150684, 2020.

[62] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Olivier De Vel, Paul Montague, and Yang Xiang. Software vulnerability discovery via learning multi-domain knowledge bases. *IEEE Transactions on Dependable and Secure Computing*, 18(5):2469–2485, 2019.

[63] Zhen Li, Jing Tang, Deqing Zou, Qian Chen, Shouhuai Xu, Chao Zhang, Yichen Li, and Hai Jin. Towards making deep learning-based vulnerability detectors robust. *arXiv preprint arXiv:2108.00669*, 2021.

[64] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering*, 47(1):67–85, 2018.

[65] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Ro{bert}a: A robustly optimized {bert} pretraining approach, 2020.

[66] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. Transformer-based language models for software vulnerability detection: Performance, model's security and platforms. *arXiv preprint arXiv:2204.03214*, 2022.

[67] Flawfinder. https://dwheeler.com/flawfinder/. 2002.

[68] RATS. Rats. 2023.

[69] Fabian Yamaguchi. Pattern-based vulnerability discovery. 2015.

[70] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, Bing Mao, and Li Xie. Autopag: towards automated software patch generation with source code root cause identification and repair. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 329–340, 2007.

[71] Jong-Deok Choi and Andreas Zeller. Isolating failure-inducing thread schedules. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 210–220, 2002.

[72] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 344–360, 2015.

[73] Carter Yagemann, Matthew Pruett, Simon P Chung, Kennon Bittick, Brendan Saltaformaggio, and Wenke Lee. Arcus: Symbolic root cause analysis of exploits in production systems. In *USENIX Security Symposium*, pages 1989–2006, 2021.

[74] Carter Yagemann, Simon P Chung, Brendan Saltaformaggio, and Wenke Lee. Automated bug hunting with data-driven symbolic root cause analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 320–336, 2021.

[75] Larissa Braz, Christian Aeberhard, Gül Çalikli, and Alberto Bacchelli. Less is more: supporting developers in vulnerability detection during code review. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1317–1329, 2022.

[76] Balázs Mosolygó, Norbert Vándor, Gábor Antal, Péter Hegedűs, and Rudolf Ferenc. Towards a prototype based explainable javascript vulnerability prediction model. In *2021 International Conference on Code Quality (ICCQ)*, pages 15–25. IEEE, 2021.

[77] Angelo Sotgiu, Maura Pintor, and Battista Biggio. Explainability-based debugging of machine learning for vulnerability discovery. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, pages 1–8, 2022.

[78] Yi Mao, Yun Li, Jiatai Sun, and Yixin Chen. Explainable software vulnerability detection based on attention-based bidirectional recurrent neural networks. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 4651–4656. IEEE, 2020.

[79] Scarlett Taviss, Steven HH Ding, Mohammad Zulkernine, Philippe Charland, and Sudipta Acharya. Asm2seq: Explainable assembly code functional summary generation for reverse engineering and vulnerability analysis. *Digital Threats: Research and Practice*, 2023.

[80] Litao Li, Steven HH Ding, Yuan Tian, Benjamin CM Fung, Philippe Charland, Weihan Ou, Leo Song, and Congwei Chen. Vulanalyzer: Explainable binary vulnerability detection with multi-task learning and attentional graph convolution. *ACM Transactions on Privacy and Security*, 26(3):1–25, 2023.

Appendix A. Survey

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

typedef int bool;

#define TRUE 0
#define FALSE 1

/**
 * User of the grocery store system.
 */
typedef struct
{
    char *username; // Username of the user.
    char *password; // Password of the user.
} User;
```

Listing 2. The included headers defined constants and defined structures used in the survey.

To assess the efficacy of our proposed system, we conducted a comprehensive evaluation involving 56 participants divided into control and assisted groups encompassing diverse educational and professional backgrounds with programming capabilities. The participants engaged in an online survey, wherein the survey's objective was presented as an evaluation of code writing quality. Initially, participants were questioned about their software development proficiency, including the extent of their coding experience and primary programming language expertise.

A.1. Survey Workflow

We initially directed participants to complete ten functions within an incomplete C source code file. These functions were intentionally crafted in a manner where potential vulnerabilities could be introduced if not approached with caution. Eligibility criteria for participation necessitated a fundamental understanding of the C programming language, ensuring that the tasks were within the participants' grasp. It is important to emphasize that the ten functions exclusively revolved around elementary C programming concepts and excluded intricate topics like data structures. In cases where participants did not fulfill any of the functions, they were gracefully guided to the survey's conclusion. This approach aimed to maintain a consistent and relevant experience for all participants. Listing 2 shows the header file and the structure we provided to the participants.

```c
void exportPrices(char **itemStrings, int numItems, char *filename)
{
    /**
     * Problem:
     *  Export the contents of the grocery store system to a file. Each line of the file should
     * contain one grocery store item.
     *
     * Example:
     *  exportPrices({"Apple - $1.25", "Orange - $0.75"}, 2, "out.txt") the contents of out.txt would
     * look like:
     * Apple - $1.25
     * Orange - $0.75
     *
     * Parameters:
     *  itemStrings: an array of grocery store items formatted as "Item - $X.XX".
     *  numItems: the length of itemStrings.
     * filename: the name of the output file to export prices to.
     */
    // —> PARTICIPANT: ENTER HERE TIME STARTED <—
}
// —> PARTICIPANT: ENTER HERE TIME COMPLETED <—
```

Listing 3. A function to write the updated price of each items to a file

Furthermore, 3 and 4 exemplify some other code samples provided to the participants. These are empty functions with a proper description of the problem, a sample test case of the solution, and a clarification on the return type. Inside the function body, there is an option to put the starting and ending times to measure how long it took them to complete the function.

If participants made partial progress on the coding assignment, a series of inquiries ensued regarding their familiarity with code vulnerabilities and the extent of their formal cybersecurity training. Following the collection of security-related background information, participants were assigned to either ”Form A” or ”Form B,” with ”Form A” representing the assisted group and ”Form B” serving as the control group. The survey structure was thoughtfully arranged such that every alternate participant received ”Form B,” a measure intended to achieve an equitable distribution of approximately 50% between the control and assisted groups.

Irrespective of the assigned form, participants were presented with five distinct C functions, which they were asked to repair. The control group only had visibility to the CWE class of the vulnerability and the vulnerable line, as depicted in Figure 1. However, the assisted group was provided with the CWE category of the vulnerability with a static description, the vulnerable line, and the root cause of the vulnerability as depicted in Figure 3.

Upon completing the vulnerability assessment task, participants were further probed with questions assessing their confidence levels in mitigating code vulnerabilities and whether they sought assistance from ChatGPT for code composition. The survey concluded with the collection of pertinent demographic data from each participant.

```
double extractPrice(char *itemString)
{
    /**
     * Problem:
     *  In the grocery store, prices are stored in the system in the format of ”Item - $X.XX”.
     * This function should extract the price from the string and return it as a double.
     *
     * Example:
     *  extractPrice(”Apple - $1.25”) returns 1.25
     *
     * Parameters:
     *  itemString: a grocery store item formatted as ”Item - $X.XX”.
     *
     * Returns:
     *  the price of the item as a double.
     */
    // —> PARTICIPANT: ENTER HERE TIME STARTED <—
    return 0.0;
}
// —> PARTICIPANT: ENTER HERE TIME COMPLETED <—
```

Listing 4. A function that primarily checks if the participant scanned values from a string correctly.