MASTER'S THESIS

Combining program slicing and graph neural networks to detect software vulnerabilities

de Kraker, WJC

Award date:
2022

Combining program slicing and graph neural networks to detect software vulnerabilities

by

Wesley de Kraker

Open Universiteit
www.ou.nl

COMBINING PROGRAM SLICING AND GRAPH NEURAL NETWORKS TO DETECT SOFTWARE VULNERABILITIES

by

Wesley de Kraker

Student number:		
Course code:	IM9906	
Degree programme:	Open University of the Netherlands, Faculty of Science, Master's Programme in Software Engineering	
Thesis committee:	Harald Vranken (supervisor)	Open University
	Arjen Hommersom (supervisor)	Open University
Thesis defense:	2022-07-20 15:45	

ACKNOWLEDGEMENTS

While writing this thesis, I have received a great deal of guidance and support. To those people, I would like to express my sincere gratitude.

First, I would like to thank my supervisors for their help in formulating the research question and methodology. Their input enabled me to find an interesting research question that has the potential to shape further research in the field of deep learning and software vulnerabilities. In addition, their critical attitude ensured that I was able to achieve the maximum results.

I would also like to thank my fellow student Davey Mathijssen. We have worked together on several assignments at the Open University and I am proud of our results. Davey started writing his research proposal at the same time as I did, and he introduced me to the field of artificial intelligence and software vulnerabilities. I cannot imagine finding a topic more interesting and educational than this.

Finally, I would like to thank my family for their continuous love and support. My parents have always encouraged me to follow my passion for software engineering, and I am forever grateful to them for giving me the opportunities and experiences that shaped who I have become. I am also grateful to my sister, who was always interested in my research efforts and helped me come up with new ideas.

CONTENTS

SUMMARY

Recent studies show that deep-learning models outperform software vulnerability detection tools that rely on human-defined rules. Deep learning focuses on neural networks that can automatically improve at a given task using data. According to prior work, a specific type of neural network is well suited for detecting software vulnerabilities. This type of network is called a graph neural network (GNN). GNN's are specifically designed to learn from graph data. Learning from this type of data is ideal for vulnerability detection because program syntax and semantics can be well expressed in graph structures. Examples of graphs that represent program syntax/semantics are abstract syntax trees, control flow graphs, and data flow graphs.

Although GNN's outperform previous approaches due to their high F1 score, there is still room for improvement. One of the areas of improvement is that these GNN studies are limited to detecting vulnerabilities at the function level. Detection at this level means that the neural network performs predictions on a function and ignores the rest of the program. As a result, these approaches are unable to detect vulnerabilities caused by the interaction of multiple functions.

In this study we investigate the effect of detecting vulnerabilities across function boundaries. To achieve this goal, we extend prior work using a technique called inter-procedural program slicing. This technique is designed to extract a subset of a potentially large program based on a specific criterion. By choosing a security criterion, slicing results in a subprogram that preserves behavior related to the existence of a vulnerability. This subprogram can consist of multiple functions, which allows the detection of vulnerabilities across function boundaries. Although program slicing has already been used for vulnerability detection, we distinguish ourselves in two ways. We are one of the first studies to combine slicing with GNN's and we included more language constructs in the slices.

To train and evaluate our deep-learning model, we collected a data set of both vulnerable and non-vulnerable C/C++ code. This data set combined samples from the Software Assurance Reference Dataset (SARD) and National Vulnerability Database (NVD). To limit the scope of our research efforts, we focus on two prevalent types of vulnerabilities: out-of-bounds writes (CWE-787) and out-of-bounds reads (CWE-125).

Using this data set, we first tried to find the optimal target depth for our program slicing algorithm. The term target depth refers to the maximum depth of the call tree. We experimented with a target depth ranging from zero to the max depth of our data set. Evaluating the accuracy and F1 score shows an increase in performance as the function depth increases. We conclude that limiting the function depth loses key information required to classify our samples.

To compare our program slicing approach with function-level analysis, we reproduced the results of a previous study and made several key enhancements to their model. These enhancements included bug fixes and a new word embedding strategy. Next, we ran the model on our new data set and compared the F1 score of both approaches. The program slicing approach performed significantly better on the data set because a third of the vulnerabilities required inter-procedural analysis. This confirmed our hypothesis that buffer vulnerabilities can occur across a series of functions and thus that program slicing outperforms function-level analysis in certain scenarios. However, this improvement in F1 score did have a drawback. Calculating a single epoch took longer, but was still feasible in a limited amount of time.

1. INTRODUCTION

1.1. SOFTWARE SECURITY

In today's society, we are dependent on software to live our daily life. Software enables online banking, controls our vehicles, and operates critical infrastructure (e.g., power plants, drinking water facilities, and fuel pipelines). Criminals are aware of this reliance on software, and they understand that traditional crime using weapons has higher risks than anonymous cyber attacks. This anonymity is one of the reasons why criminals are shifting from traditional crime to cyber attacks [47].

An example of such an attack took place at Colonial Pipeline, a company that operates the largest fuel pipeline in the United States. On April 29, 2021, cybercriminals gained access to the network of this company [48]. About a week later, the criminals stole nearly 100GB of data and encrypted company files. The cybercrime group called DarkSide, demanded more than 4 million dollars in cryptocurrency to decrypt the files and prevent leakage of the stolen data. In response, the company decided to pay the ransom and shut down the pipeline for almost a week. After examining the pipeline (29,000 miles long) and computer networks, the company resumed service. The shutdown led to gas shortages, higher prices at the pump, and president Biden declaring a regional emergency [35].

Software security is about building applications that can withstand attacks [15]. Criminals launch attacks to steal data or disrupt business operations, as seen in the Colonial Pipeline attack. Software defects with security ramifications enable these attacks. These defects include implementation bugs such as buffer overflows and design flaws such as the use of single-factor authentication. By exploiting defects, a cybercriminal can perform unintended actions on an application. In the case of Colonial Pipeline, the lack of multi-factor authentication allowed the criminals to log in to the network with stolen credentials.

1.2. VULNERABILITY DETECTION

A software defect with security ramifications is also known as a *software vulnerability* [15]. To prevent cybercriminals from exploiting software vulnerabilities, organizations need to find and patch them before the criminals exploit them. Vulnerability detection is the process of finding these vulnerabilities. Organizations often employ *vulnerability detection tools* to speed up this otherwise manual process.

These tools automatically detect vulnerabilities using static analysis, dynamic analysis, or both [4]. Static analysis is performed without executing a program. Vulnerabilities are discovered by analyzing the source code or binaries of a program. Dynamic analysis uses the opposite approach and executes a program to find vulnerabilities. This approach executes an attack against the program and analyzes the response to determine if the exploit was successful.

A static code analyzer relies on human experts to define rules to distinguish between vulnerable and non-vulnerable code [4]. The advantage of this automated approach is that vulnerabilities are discovered quickly. However, the disadvantage is that these analyzers often have a high false-positive or false-negative ratio. The root cause of these incorrect classifications are the underlying rules. These rules do not capture all the ways code can be vulnerable. According to Li et al. [12], the widely used commercial product Checkmarx has both a false-positive and false-negative ratio of more than 40%.

Recent studies show that *deep learning* can overcome this challenge by automatically extracting high-quality patterns from vulnerable code samples [12, 20, 22, 29]. Deep learning is a subfield of machine learning that focuses on *neural networks*. A neural network is a set of algorithms inspired by the neurons in our brain. These networks enable computers to become better at a task (e.g. vulnerability detection) by learning from data.

According to Zhou et al. [29] and Wang et al. [22], a specific type of neural network is well suited for detecting vulnerabilities. This type of network is called a graph neural network (GNN). GNN's are specifically designed to learn from graph data. Learning from this type of data is ideal for vulnerability detection because program syntax and semantics can be well expressed in graph structures. Examples of graphs that represent program syntax/semantics are abstract syntax trees, control flow graphs, and data flow graphs. By combining these representations, a neural network can reason about the security properties of a program by detecting patterns in its structure, control flow, and data flow. Alternative approaches that analyze code as a sequence of tokens fail to take advantage of the well-defined semantics of a program [22].

Despite these advances in GNN's, there is still room for improvement. A study by Zhou et al. [29] reports an accuracy of 76% for vulnerability detection, and a more recent study by Wang et al. [22] reports an accuracy of 92%. Although this is a big step forward, in practice this still results in false-positives and false-negatives. One of the areas for improvement is that these studies [22, 29] are limited to detecting vulnerabilities at the function-level. Detection at this level means that the neural network performs predictions on a function and ignores the rest of the program. As a result, these approaches are unable to detect vulnerabilities caused by the interaction of multiple functions.

1.3. GOAL

This project aims to improve previous approaches by creating a GNN that detects vulnerabilities across a series of functions. To achieve this goal, the function-level approach of Wang et al. [22] is extended using a technique called *program slicing*. Wang et al. refer to their work as FUNDED, we will use this term hereafter.

The technique program slicing is designed to extract a subset of a potentially large program based on a specific criterion [23]. By choosing a security criterion, slicing results in a subprogram that preserves behavior related to the existence of a vulnerability [12]. This subprogram, also known as slice, is used as input for the neural network. A slice can consist of program statements across different functions [18]. This enables detecting vulnerabilities caused by the interaction of multiple functions. The advantages of program slicing are further elaborated in subsection 2.3.

Using program slicing for vulnerability detection is not new, but we are one of the first studies to combine slicing with graph neural networks. In addition to the main goal, we also improve the word embedding strategy used by FUNDED and include more language constructs in the program slicing algorithm compared to previous studies.

1.4. REPORT STRUCTURE

The report is structured in eight sections. After the introduction, we describe the research context which specifies the research area explored. In the third section the research design is outlined which includes the research motivation, goal, questions, and method. Section four describes the implementation of the data set, deep-learning model, and program slicing algorithm. After describing the implementation, we analyze the results of our experiments. Next, we compare our work with related work in this area. In section seven, we discuss the relevance of our experimental results. After discussing the relevance, we summarize the results in the conclusion. Finally, we describe the next steps that should be taken to further advance this research field.

2. RESEARCH CONTEXT

This section describes the research areas that are explored. First, we introduce the vulnerability types analyzed, namely buffer vulnerabilities. Next, different graph structures are discussed to represent (non-)vulnerable programs. These structures include the abstract syntax tree, control flow, and data flow graph. Subsequently, the program slicing algorithm is described that can extract a subset of a larger program (also known as slice). These slices are fed into a deep-learning model discussed in the penultimate section. After introducing the research areas, the related work is described.

2.1. APPLICATION SECURITY

The specific area within the software security field that we research are *applications written in C or C++*. C/C++ applications are especially vulnerable to attacks as these languages allow arbitrary memory access. This access to memory enables the development of highly performant and efficient applications but can also introduce memory safety vulnerabilities. An example of such a vulnerability is writing data past the end of a buffer.

According to MITRE [44], writing data outside the boundaries of a buffer is the number one most critical vulnerability in 2021. Reading data outside the boundaries of a buffer is ranked third on the list. Since these buffer vulnerabilities are still so prevalent, we decided to focus our research efforts on these two types of vulnerabilities. The Common Weakness Enumeration system (CWE)[36] is used to define these types and their subcategories. The CWE system is a database of software vulnerability types and definitions. The database is maintained by MITRE and it is often used as a common language for vulnerability identification.

- Out-of-bounds Write (CWE-787): a program writes data past the end or before the start of a buffer.
  - Stack-based Buffer Overflow (CWE-121): the buffer overwritten is allocated on the stack.
  - Heap-based Buffer Overflow (CWE-122): the buffer overwritten is allocated on the heap.
  - Write-what-where (CWE-123): a buffer is overwritten with an arbitrary value controlled by the attacker.
  - Buffer Underwrite (CWE-124): data is written before the start of a buffer.
- Out-of-bounds Read (CWE-125): a program reads data past the end or before the start of a buffer.
  - Buffer Over-read (CWE-126): data is read past the end of a buffer.
  - Buffer Under-read (CWE-127): data is read before the start of a buffer.

In the subsections below, we take a closer look at these two vulnerability types. Next, we discuss why not all languages are memory safe, preventing these types of vulnerabilities.

Out-of-bounds Write

As defined above, an out-of-bounds write occurs when a program writes data past the end or before the start of a buffer (CWE-787)[43]. A buffer is a piece of memory where data is temporarily stored. To temporarily store data, a specific region of memory is reserved by the buffer. Altering data outside the boundaries of a buffer changes other regions of memory and could affect a program's code or data. The consequences of altering other regions of memory include corruption of data, a crash, or the execution of malicious code. The term buffer overflow (CWE-121/CWE-122) means altering data past the end of a buffer and a buffer underwrite/underflow (CWE-124) means altering data before the start of a buffer [39].

Out-of-bounds writes can be further divided based on the memory area where the buffer is located. These memory areas are the stack (CWE-121) and the heap (CWE-122). The stack is a fixed size memory region where local variables, arguments and return addresses are stored. A return address determines the next instruction that must be executed after a function has finished. If an attacker can overwrite the return address to point to their malicious code (CWE-123), a buffer overflow vulnerability results in arbitrary code execution [37].

The heap is a memory region that can grow dynamically. This memory region is used to allocate data when the size is unknown at compile-time or if a large amount of data needs to be stored. In case a function pointer is stored on the heap, an attacker can overwrite this function pointer to refer to their malicious code (CWE-123). In this way, a heap-based buffer overflow can result in arbitrary code execution [38].

Figure 1 shows a C program containing a heap-based buffer overflow vulnerability. A destination buffer (of size 50) is created on the heap and a source buffer (of size 100) is created on the stack. The content of the source buffer is copied into the destination buffer using strncpy. However, the destination buffer (located on the heap) is too small causing a heap-based buffer overflow (50 < 100).

```c
void main()
{
    char* dest = (char*) malloc(50*sizeof(char));

    char source[100];
    memset(source, 'C', 100-1);
    source[100-1] = '\0';

    // Heap-based buffer overflow
    strncpy(dest, source, 100*sizeof(char));

    printf("%s", dest);
}
```

Figure 1: C program containing a heap-based buffer overflow vulnerability. The destination buffer (located on the heap) is too small causing a heap-based buffer overflow (50 < 100).

Out-of-bounds Read

As defined above, an out-of-bounds read occurs when a program reads data past the end or before the start of a buffer (CWE-125) [40]. Reading data outside the boundaries of a buffer can allow an attacker to extract sensitive information from other memory locations. The term buffer over-read (CWE-126) means reading data past the end of a buffer [41] and a buffer under-read (CWE-127) means reading data prior to the targeted buffer [42]. An buffer over-read occurs when a pointer/index is incremented beyond the maximum capacity. On the other hand, an under-read typically occurs when a pointer/index is decremented or when a negative index is used. In the same way as an out-of-bounds write, this vulnerability can occur on both the stack and heap depending on the location of the buffer.

Figure 2 shows a C program containing a buffer over-read vulnerability. Memory past the end of the source buffer is read because the loop ranges from 0 to 99 (string length of dest buffer). This range exceeds the source buffer's maximum capacity of 50.

```c
void main()
{
    char* source = (char*) malloc(50*sizeof(char));
    memset(source, 'A', 50-1);
    source[50-1] = '\0';

    char dest[100];
    memset(source, 'A', 100-1);
    source[100-1] = '\0';

    for (int i = 0; i < strlen(dest); i++)
    {
        // Buffer over-read
        dest[i] = source[i];
    }
}
```

Figure 2: C program containing a buffer over-read vulnerability. Memory past the end of the source buffer is read because the loop's range is larger than the source buffer's maximum capacity.

Memory safety vulnerabilities

Memory safety is the state of being protected against these buffer related vulnerabilities. A language that protects against these vulnerability types is called memory safe [7]. An example of a memory safe programming language is Java. However, in order to provide this safety the runtime environment must perform checks. These checks include validating the bounds of an array, type conversions and null pointer references. The execution overhead of these runtime checks ranges from 130% to 540% [3].

As previously described, the programming languages C and C++ allow arbitrary memory access and thus are not memory safe. This lack of safety has the disadvantage of potentially leaking information or corrupting memory but also has advantages in terms of performance and efficiency. Especially in the context of embedded applications, a common requirement is that a program must operate under low power or memory constraints. This makes it unlikely that languages like C and C++ will disappear and thus emphasizes the importance of detecting these types of vulnerabilities early.

2.2. GRAPHS

As described earlier, Wang et al. [22] used graph neural networks to predict vulnerabilities. These neural networks are designed to learn from graph data. Learning from this type of data is ideal for vulnerability detection because program syntax/semantics can be well expressed in graph structures. In the next subsections, three different graph representations for programs are discussed. These representations used by Wang et al. are the abstract syntax tree, control flow graph, and data flow graph.

Before we dive into these graphs, we first define what a graph is. A graph holds two types of information: *nodes*, and *edges*. Nodes represent the entities, and edges model the relationship between nodes. These nodes are sometimes referred to as points or vertices, and edges are the lines that connect two nodes. More formally, a graph G is defined as a pair (N, E), where N is a set of nodes and E a set of edges. The set of edges is a subset of the Cartesian product of two nodes $E \subseteq \{(n, m) | n \in N \land m \in N\}$.

The graphs described below are both directed and labeled. In a directed graph, the edges have a direction which is often indicated by an arrow. In a labeled graph the nodes are assigned a value from a set. This value is known as the node's label and does not necessarily have to be unique for the entire graph. In other words, multiple nodes are allowed to have the same label.

ABSTRACT SYNTAX TREE

Abstract syntax trees (AST) are widely used for compilers to represent the structure of a program. Other use cases include program analysis and transformation. FUNDED uses this tree structure for program analysis.

The AST is created in two phases [17]. The first phase is lexical analysis which converts a program to tokens. A token is the smallest building block of a program. There are five types of tokens: identifiers (buf, i), keywords (if, while, return), strings ("abc"), operators (+, -, /), constants (10, 'a'), and special characters (";", ",", "(", ")").

The second phase is syntax analysis (also known as parsing). This phase takes as input the sequence of tokens and converts it into a tree structure. A tree is similar to a graph but has a root node and no cycles. The result of this phase can be either a parse tree or an AST. A parse tree retains all information from the original program including special characters. The AST is a higher level representation that omits special characters.

Figure 3 shows an example for the algorithmic expression: $9 * (8 - 7)$. The tree on the left represents a parse tree and the tree on the right an AST. In the AST, the parentheses are omitted from the expression because the evaluation order can be derived from the tree. Omitting special characters, such as parentheses, simplifies the tree structure without losing semantic information.

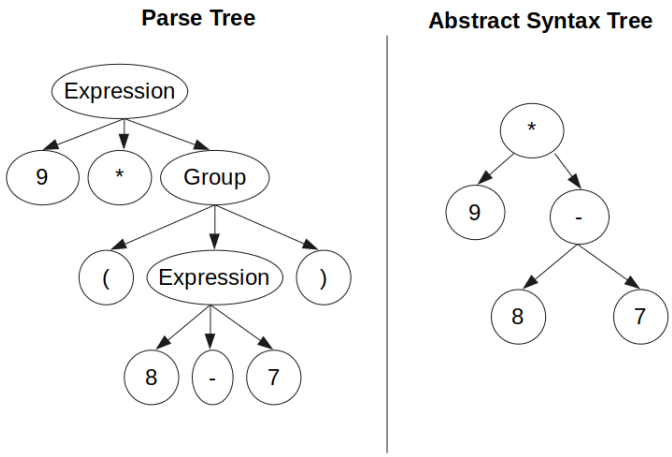

Figure 3: Parse Tree and Abstract Syntax Tree for the expression: $9 * (8 - 7)$.

In an AST, the internal nodes represent operators. The leaf nodes represent operands. An operator indicates the actions to be performed. For example, arithmetic operators are add, subtract, and divide. The operands are the expressions where the action is applied on (e.g. identifiers/constants).

CONTROL FLOW GRAPH

The control flow graph (CFG) represents all paths that can be traversed during program execution. This graph is commonly used in the field of static analysis and compiler optimization. It can address questions such as: is this code reachable during execution or is it unreachable code? Could this statement be executed multiple times? The control flow graph was introduced by Frances E. Allen in 1970 [1]. Her work was inspired by the boolean connectivity matrices of Reese T. Prosser [19].

A control flow graph is a directed graph. The nodes in this graph represent basic blocks and the edges represent control flow. A basic block is a list of program statements that are executed from top to bottom without branches. Control flow statements alter this flow using decision-making statements, looping and branching. Decision-making statements (e.g. if, else, and switch) are used to execute a basic block if a specified condition is met. Looping statements (e.g. for, while, and do-while) allow the repetition of a basic block for a specified number of times. Branching statements end the execution of a loop or function (e.g. break, continue).

Figure 4 shows the control flow graph for several control flow statements. In the if statement, the node on the left is only executed if the condition evaluates to true. All looping statements contain cycles which means that there is a path where the first and last node are equal. Note that the while and for loop are equivalent because both control flow statements iteratively execute a block of code if the condition evaluates to true. The difference between the while and do-while loop is that the do-while executes the block of code at least once, even if the condition evaluates to false.

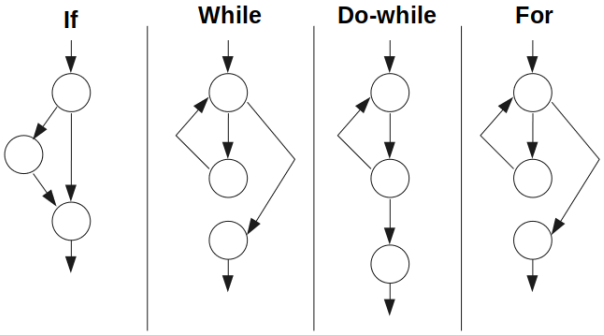

Figure 4: Control flow graph of several control flow statements.

DATA FLOW GRAPH

A data flow graph (DFG) represents how data is passed through the program. It is also used in the field of static analysis and compiler optimization. It can address questions, such as: where does the value of this variable come from? Is this variable initialized before use?

Data flow is captured in data dependence edges. To compute these edges, we use the algorithm proposed by Frances E. Allen [2]. A data dependence edge connects definition-use pairs. A variable definition for a variable v is a statement that assigns to v. This variable v can be a local, global variable, parameter or field. A variable use for a variable v is an expression that reads the value of v. This variable may be read in a condition or as part of a computation. A formal definition of a definition-use pair and data dependence edge is given below:

Definition 1: a *definition-use pair* for a variable v is a pair of nodes (d,u) such that there is a definition-clear path from d to u in the control flow graph. A definition clear path is a path $n_1,...,n_k$ in the control flow graph (CFG) such that n_1 is a variable definition and n_k is a variable use. No node in between n_1 and n_k is allowed to be a variable definition for v.

Definition 2: a *data dependence edge* connects a definition d with its use u according to the previous definition.

Figure 5 shows the data flow graph for a function that calculates the power of two. Below the program, the data dependence edges are written down as (n,m) where n is the definition and m is the use. Next to the program, a control flow graph is given. This control flow graph can be used to verify that there exists a control flow path between the definition and use.

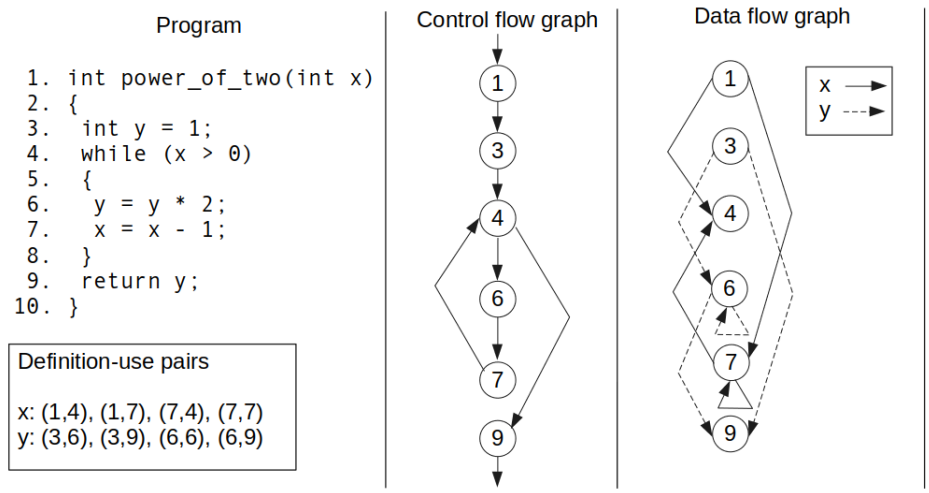

Figure 5: Data flow graph capturing the definition-use pairs.

2.3. PROGRAM SLICING

Mark Weiser first introduced program slicing in 1984. He described it as a method for automatically creating a subset of a program that preserves certain behavior [23]. This is achieved by extracting program statements that influence the values at a particular location. The *slicing criterion* determines which values at a certain location we are interested in. An observer only looking at the values (of the slicing criterion) should be unable to distinguish a run of the program from a run of the slice (subset of the program). The values of both runs should be the same.

There a various applications for program slicing according to Mark Weiser [23]:

- Debugging: find parts of a program that are relevant for a bug;
- Program understanding: understand which statements affect a particular variable;
- Parallelization: decide which parts of a program can be computed independently from each other.

We could argue that vulnerabilities are a type of bug and thus that vulnerability detection is a similar application as debugging. This reasoning is inspired by the authors of VulDeePecker/SySeVR [12, 13], who were the first to combine program slicing and deep learning to find software vulnerabilities. They proposed using commonly misused library functions as the slicing criterion. Based on root cause analysis of the vulnerabilities in our data set, we used a similar strategy. This strategy is discussed later in subsection 4.1.

Horwitz et al. [9] extend the approach of Mark Weiser by introducing a new type of graph, called *system dependence graph*. This graph extends the previous representation by including the interaction between functions. Capturing the calls between functions in a program is also known as *inter-procedural analysis*.

Performing deep learning on an inter-procedural program slice instead of a function has two advantages. As mentioned earlier, the first advantage is that vulnerabilities can be discovered that are caused by the interaction of multiple functions. For example, whether a function is vulnerable may depend on the arguments passed to it. In that case, it is crucial to analyze both the caller and callee. Analyzing the caller function gives insight into the actual arguments passed to a function, and analyzing the callee reveals further usage of these arguments.

The second advantage of slicing is that security-irrelevant program statements are excluded. This is enabled by the security criterion used to slice the program. A security criterion guarantees that only relevant statements to vulnerability detection are included in the resulting subprogram. Zhou et al. [29] hypothesize that this enables more efficient learning for a program with large functions.

EXAMPLE PROGRAM SLICING

Figure 6 shows an example of vulnerable C code that would benefit from program slicing. The example code is vulnerable because data is written past the end of a buffer. Program execution starts in the "main" function. At line 10, a buffer is created of size ten and assigned to the variable source. The buffer is filled at line 11 and 12 with nine B's and a null terminator: "BBBBBBBBB\0". The null terminator is required to indicate where the string ends. The buffer of size ten is passed to the function "Print_With_A_Prefix". This function creates a buffer of size five and sets the first character to "A". The buffer of size five is assigned to the variable dest (shorthand for destination). After which, the "strcat" function tries to append a copy of the source buffer to the destination buffer. The problem is that this string "ABBBBBBBBB\0" is eleven characters long. The destination buffer has a max capacity of five characters, which causes data to be written past the end of the buffer. This is called a buffer overflow.

In addition to the C code, the corresponding control flow graph is also shown. Statements that are relevant to the existence of the vulnerability are marked green. Line 16 and 18 are irrelevant because they do not affect or use the values passed to the function "strcat". These lines only print the sequence "0 1 2 3 4 5 6 7 8 9 ".

Figure 7 illustrates the difference between a function-level approach and a program slice. On the left the control flow graph is split by function, and on the right a program slice is shown. The function-level approach splits the graph in two because the "main" function and "Print_With_A_Prefix" function are analyzed in isolation. The "main" function is defined as lines 10 through 19 and the "Print_With_A_Prefix" function is defined as lines 3 through 5. The disadvantage of this approach is that the neural network is either fed a graph with information about the source or destination buffer size but not both. As a result, the neural network cannot determine whether a buffer overflow will occur and thus whether the code is vulnerable.

On the right a program slice is shown, the neural network can determine whether the code is vulnerable because the graph is not split in two. The security-related function call "strcat" is marked in this example because it was used as the slicing criterion. Moreover, the security-irrelevant lines 16 and 18 were excluded because they do not influence the value passed to the "strcat" function.

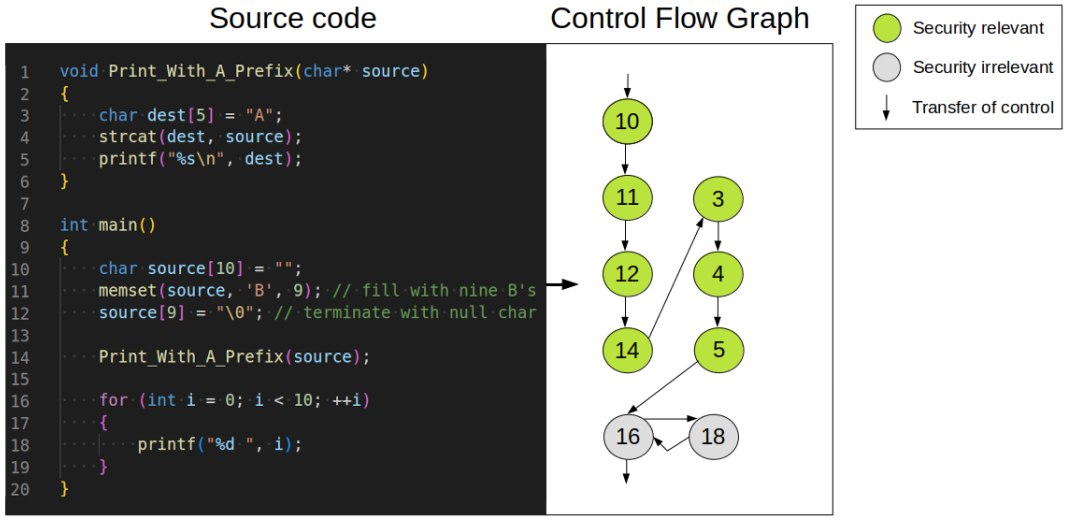

Figure 6: C program containing a buffer overflow vulnerability and its corresponding control flow graph. Statements in the graph that are relevant to the existence of this vulnerability are marked green.

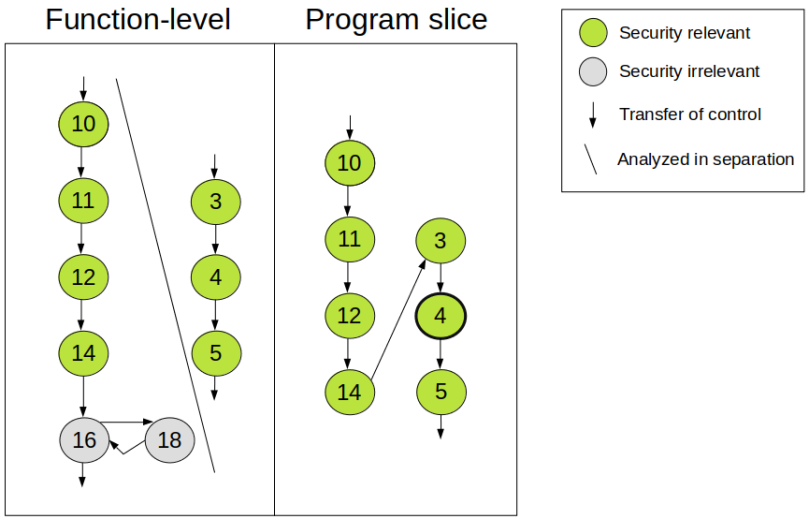

Figure 7: Function-level vs program slice (of "strcat" function).
The program slice is superior because it includes additional calling context and excludes security-irrelevant statements. In the function-level approach, either the size of the source or destination buffer is unknown.

2.4. SLICING ALGORITHM

There are two methods to implement the program slicing algorithm: static or dynamic. The static method analyzes the source code without executing it. The dynamic method executes the program for a particular input and computes the slice based on the execution path. We chose the static method because dynamic slicing only considers a particular set of executions. Moreover, automatically compiling and running real-world software projects is not easy because developers use a wide range of build systems for C/C++ programs (Make, CMake, Ninja, SCons, Meson, among others).

In addition to static versus dynamic slicing, a distinction can also be made between backward and forward slicing. Backward slicing analyzes statements that affect the slicing criterion and forward slicing analyzes the statements that are influenced by a particular criterion. We chose backward slicing because we are interested in statements that affect the arguments of a security-related library function call.

The slicing algorithm used for our implementation extends the original algorithm described by Mark Weiser [23]. Weiser's algorithm reduces the problem to a graph reachability problem based on the program dependence graph. The approach does this by computing the graph from the source code and determining which nodes are reachable by the target nodes. These target nodes are defined by the slicing criterion. The algorithm of VulDeePecker and SySeVR uses commonly misused library functions as the criterion.

A program dependence graph is a directed graph representing the data and control dependence between statements/expressions. The nodes in the graph represent the statements or expressions. The edges represent either data dependence or control dependence. A definition for data dependence is already given in subsubsection 2.2.3. Control dependence is defined below. Since post-domination is part of the control dependence definition, we introduce this term first.

Definition 3: Node n_2 *post-dominates* node n_1 if every path $n_1, ..., exit$ in the control flow graph contains n_2. In other words, in order to get out of n_1 we have to go through n_2.

Definition 4: *control dependence* means that a statement n_2 is executed depending on the outcome of an expression at statement n_1. Statement n_2 is control dependent on n_1 if the following two criteria are met:

- There exists a control flow path $P = n_1, ..., n_2$ where n_2 post dominates every node except n_1 in P;
- n_2 does not post-dominate n_1.

For computing the slices, we first need to compute the data/control dependence edges. Next, we define the slicing criterion (n, V), where n is a statement and V is the set of variables used at n. In our algorithm the statement is a security-related library function call (defined in Table 3). The slice for (n, V) are all statements reachable from the set V at n. In other words, the slice is equal to all statements on which the variables V at a particular statement n depend.

The above definitions are extended using the work of Horwitz et al. [9]. As previously mentioned, they introduced the system dependence graph as an extension of the program dependence graph. The system dependence graph also captures calls between functions. This means that the data dependence edges also connect arguments and parameters. By computing the graph reachability problem over this graph, the calling context of a function is also included if data/control dependent. We use their extended graph in our implementation.

The pseudocode below implements a solution for the graph reachability problem. Based on the slicing criterion, we iteratively find nodes reachable through a control or data dependence edge. The result is a set of nodes S representing the program slice.

$C \leftarrow \{...\}$, set of nodes defined by the slicing criterion
$T \leftarrow C$, set of nodes to be visited (starting with the slicing criterion C)
$S \leftarrow \{\}$, set of nodes representing the program slice

while $|T| > 0$ **do**
    $n \leftarrow Pop(T)$, remove first element from the set and assign it to n
    if $n \notin S$ **do**
        $S \leftarrow n \cup S$
        $R \leftarrow ReachableBy(n)$, find nodes reachable by control/data dependence
        $T \leftarrow R \cup T$
    end
end

2.5. DEEP LEARNING

Deep learning is used to automatically distinguish vulnerable from non-vulnerable C/C++ code because according to Li et al. [12] it can outperform other methods. This section introduces what deep learning is and what problems it can solve.

Deep learning is a subfield of machine learning that focuses on *artificial neural networks*. An artificial neural network is a set of algorithms inspired by the neurons in our brain. The human brain consists of a massive amount of interconnected neurons that allow us to learn and generalize.

Figure 8 shows an example of such a biological neuron. A neuron is a biological cell that processes information [10]. It consists of a cell body and two types of branches called dendrites and axon. The dendrites receives electrical signals from other neurons. Based on these input signals, the cell body can generate new signals which are sent across the axon. A synapse connects this axon to the dendrites of another neuron. The amount of electrical signal that arrives at the other neuron depends on the strength of the synapse. This strength can be adjusted by the signals passing through which enables learning.

The human brain consists of 10^{11} neurons that are highly interconnected [10]. The electric signals that neurons use to communicate typically last for a couple of milliseconds. This means that the human brain can transmit signals at a rate of a few hundred hertz (signals per second). This speed is extremely slow compared to modern CPU's. For example, the CPU used to write this paragraph operates at a speed of 3.6×10^9 hertz. However, our brain is still able to perform complex tasks such as face recognition in a relatively short amount of time. This implies that our brains processes input in parallel to achieve this level of performance.

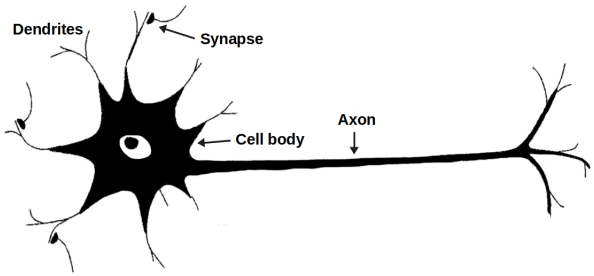

Figure 8: Biological neuron [10].

ARTIFICIAL NEURAL NETWORKS

In 1943, McCulloch and Pitts introduced the term artificial neuron [14]. An artificial neuron is a computational model inspired by the biological neurons in our brain (see Figure 9). This computational model receives input signals (as a numeric value), sums these signals, and then produces its own output. The inputs represent the dendrites and the outputs represents the axons. Each input is weighted before being summed, this mimics the strength of the synapse. More formally, given a neuron, let there be n inputs with signals x_j and weights w_j. The transfer function φ maps each possible sum to an output (usually a value between 0 to 1).

$$y = \varphi\left(\sum_{j=1}^{n} w_j x_j\right)$$

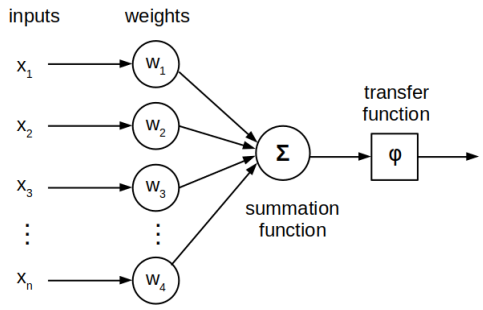

Figure 9: Artificial neuron [10].

An artificial neural network can be seen as a graph in which nodes represent neurons and edges represent the connections and weights between neurons. In their work, McCulloch and Pitts proved that a neural network with correctly adjusted weights can perform any computation. Learning is the process of updating these weights to become better at a certain task.

The architecture of a neural network can be divided in two categories: feed-forward neural networks and recurrent neural networks. Each architecture consists of at least one input layer, one or more hidden layers, and one output layer. The neurons in the input layer represent a feature from our data set. The hidden layers are located between the input and output and perform transformations. The neurons in the output layer produce the final prediction, for example vulnerable or non-vulnerable.

Figure 10 shows a typical network for each architecture. The feed-forward network has by definition no loops. Signals travel from input to output. In other words, the output of any layer only affects subsequent layers. A feed-forward network is also memory-less meaning the output is independent of previous data fed into the network. In a recurrent network, signals can travel in both directions. The self-loops added in Figure 10 enable the network to store previous data. This information is fed back into the network once it receives new data. By feeding information back from the previous step, the network has a form of memory.

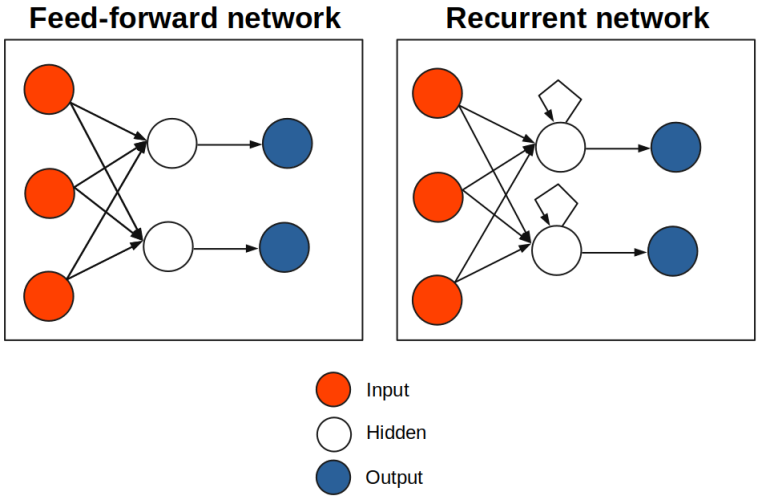

Figure 10: Feed-forward vs recurrent neural network.

LEARNING

As mentioned earlier, learning is the process of updating weights to become better at a certain task. However, manually adjusting weights in an artificial neural network would be a time-consuming task. To automate this process, algorithms exist that adjust the weights to obtain the desired result. Examples of the desired result could be recognizing objects in an image or classifying whether software code is vulnerable.

There are two main learning paradigms: supervised and unsupervised learning [10]. Supervised learning relies on labeled input data. In other words, the expected answers (output) of the network must be known. Weights are updated to produce results as close to the expected output as possible. Unsupervised learning does not require labeled data. This learning method finds patterns in the input data and uses these patterns to organize it into categories.

In this work, we focus on supervised learning for multi-layer neural networks. To automatically adjust weights, we need two algorithms: a back-propagation and an optimization algorithm. Initially, the weights of the neural network are set to random values. Next, the neural network is fed input data and this input is passed through the network (forward pass). By comparing the predicted output with the desired output, the loss is calculated. The loss is the distance between the network's output and the expected output. There are several functions to calculate the loss, such as the cross-entropy loss function or the mean squared error.

After calculating the loss, back-propagation is used to compute the gradient with respect to the weights. A gradient captures the slope of the loss function. This slope allows us to predict how a weight should be updated to minimize loss. In mathematical terms, the gradient measures how a change to the inputs (weights) affects a function's output (loss). Figure 11 shows how the gradient can be used to minimize loss of a random weight. Optimizers (such as stochastic gradient descent and adam) aim to iteratively update the weights to reach the global minimum. The global minimum is the point where the loss is the smallest.

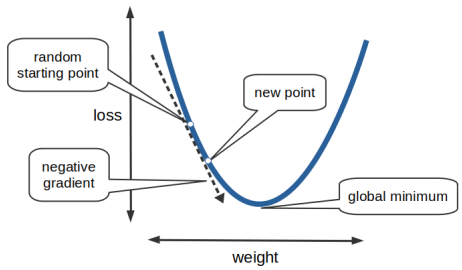

Figure 11: Changing the weight in the direction of the negative gradient reduces loss.

Most of the time, the network does not reach an optimal loss after a single pass through the data set. The network often has to "see" the data multiple times. A single pass through the data set is known as an epoch. The number of epochs required to reach optimal loss depends on multiple factors including the learning rate. The learning rate is a parameter of the optimizer and determines how drastically the weights change. A learning rate that is too high may risk the possibility of overshooting. This occurs when the network updates the weights too much, causing the network to go past the point of minimized loss. On the other hand, a too low learning rate requires much more training time to reach optimal loss.

Evaluation

After training the neural network, we want to evaluate how the network performs on unseen data. To achieve this goal, we split the data in a data set for training, validation, and testing. The training set is used during the learning phase to update the weights of the neural network. The goal is that when we deploy the network in production, it is able to accurately predict data it has not seen before. The neural network makes these predictions based on what it learns from the training set.

The validation set is used to evaluate the network during training and is separate from the training set. The validation set helps us in choosing the optimal hyperparameters (such as the number of epochs). After each epoch, the network is evaluated on this data set. The main difference with the training set is that this set is not used to update the weights. Evaluating the network on the validation set can prevent overfitting.

The test set is used to evaluate our model after training. It provides an unbiased estimate of the performance of the final model. This set is separate from both the training and validation set.

Overfitting/underfitting

Two common issues can arise when the neural network is learning: overfitting and underfitting. Overfitting occurs when the model becomes good at classifying data in the training set, but performs poorly on the test set. By introducing a validation set, we can see at which epoch a model starts to overfit. A model starts to overfit, if the validation metric is considerably worse than the training metric. There are several ways to prevent overfitting: adding more data, reducing the complexity of the neural network (less layers/neurons), and using regularization techniques.

Underfitting occurs when the neural network cannot find patterns in the input data that produce the correct output. The result is a poor performance on the training set. Poor performance is indicated by a high loss or low accuracy. If the model cannot accurately classify data it has been trained on, it is unlikely to generalize to unseen data. There are several ways to reduce underfitting: increase the complexity of the neural network and reduce regularization techniques.

Regularization is a technique that reduces overfitting by penalizing for complexity. By trading in some of the ability of the neural network to fit the training data, the network becomes better at generalizing to unseen data. Two techniques commonly used to achieve this are: L2 Regularization and dropout. L2 Regularization penalizes for large weights in the neural network. Dropout randomly ignores certain neurons in the network.

2.6. GRAPH NEURAL NETWORKS

The model FUNDED uses a specific type of GNN to detect vulnerabilities, called Gated Graph Neural Networks (GGNN) [22]. This subsection introduces the inner workings of a GGNN.

A GNN, as the name suggests, is a neural network designed to learn from graph structures. The concept of GNNs is relatively new. For a long time, neural networks could only operate on sequences and grids. This is not a problem for images that can be represented as grids or text/audio that can be represented as a sequence of information.

However, problems start to arise when neural networks need to learn from control/data flow graphs, social network graphs [24] or molecular structures [8]. These structures cannot be directly mapped to a grid or sequence. The reason is that graphs often have an arbitrary size and lack of ordering. The arbitrary size makes it hard to map a graph to a fixed-size grid. The lack of order makes it hard to determine the order of nodes in a grid or sequence.

GNNs solve this problem by enabling learning on this data structure [21]. Learning is enabled by algorithms that map graphs into an *embedding space*. In this context, an embedding space refers to a lower-dimensional representation of high-dimensional graph data. This lower-dimensional representation can be used to distinguish graphs. Figure 12 shows an example of how graphs are mapped to two-dimensional space. Graphs that are similar in structure are close to each other. The top two yellow graphs only differ in orientation and thus are mapped to the same point. By splitting the two-dimensional space, classes can be assigned to the graphs. For example, the yellow graphs may represent vulnerable code, and the blue graphs non-vulnerable code.

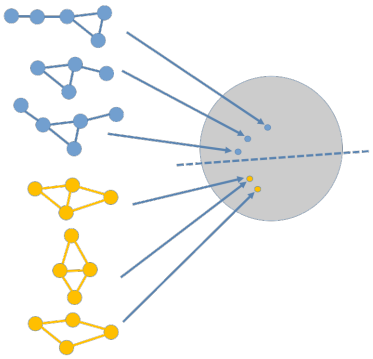

Figure 12: Graphs mapped into embedding space based on similarity in structure [31].

The task previously described of classifying graphs is also known as graph-level classification. There are other tasks that a GNN can perform but we focus on this particular task because the FUNDED model utilizes it [22]. Graph-level classification means that the input to the network is a graph. Based on this input graph the network determines the class it belongs to. In the case of FUNDED, the input graph represents a function of a program. In their work this graph is a combination of the abstract syntax tree (AST), control flow, data flow, and control dependence graph. This combined graph determines the class predicted by the network which can be either vulnerable or non-vulnerable.

A graph neural network maps graphs into a lower-dimensional representation using two steps: propagation and pooling [28]. The propagation module transfers information between nodes so that a node captures both its own features and how it relates to neighbors. Next, the pooling module combines all nodes to create the final graph representation.

Propagation is achieved by transforming a node's representation using the neighboring nodes. A node is a neighbor if it is connected to the target node by an edge. All the neighbor's sent their representation to the target node using messages. After aggregating the messages, the target node is updated using this aggregated information. This propagation procedure occurs for all nodes and often occurs multiple times to enable the exchange of messages beyond neighbors.

In the second step the nodes are pooled to create a graph representation. This pooling operation occurs after the nodes are updated in the propagation step. A simple example of a pooling operation would be to sum all nodes. This results in a fixed-sized representation of the graph. The fixed-size representation can be used as input for a classifier. Figure 13 shows an example of how propagation and pooling work. The node representations are visualized using colors.

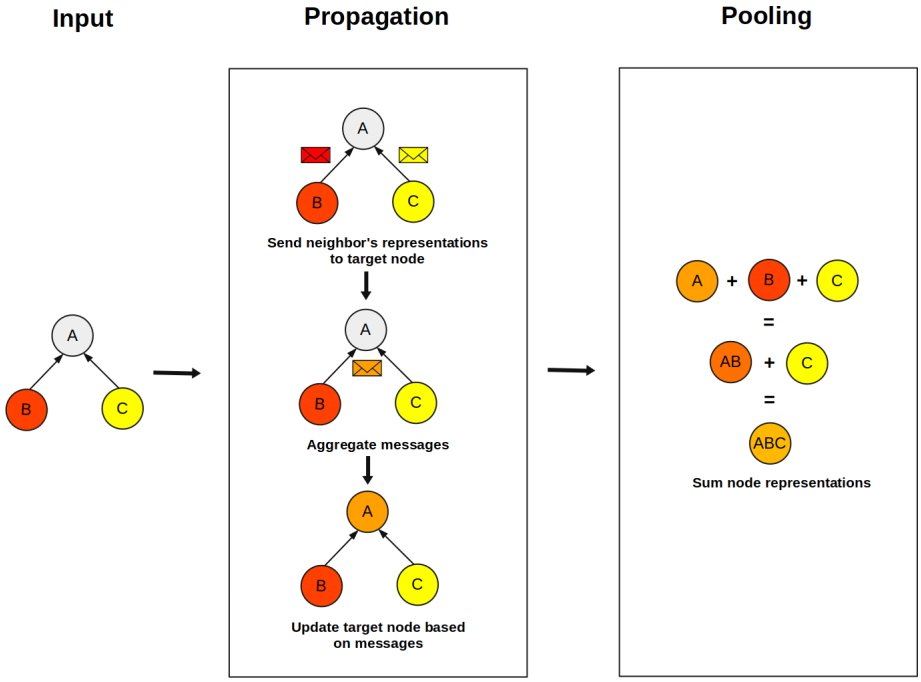

Figure 13: Convert graph to a lower-dimensional representation using propagation and pooling.

2.7. VULNERABILITY DETECTION

To understand why we chose to extend FUNDED [22] using program slicing, we first need to introduce prior research.

VULDEEPECKER

One of the first studies to combine deep learning and vulnerability detection was VulDeePecker [12]. Their approach combined data flow analysis and neural networks that can cope with sequential data. First, they performed program slicing on security-related library functions. The resulting slices were transformed into sequences of tokens and fed into a BLSTM (Bidirectional Long Short-Term Memory) neural network. This type of recurrent neural network can process sequences of tokens by storing information about previously seen tokens in memory.

SYSEVR

A follow-up study called SySeVR improves on several of VulDeePecker's shortcomings [13]. Largely the same research team as VulDeePecker conducted this follow-up study. One of the improvements was extending the data flow analysis with control flow analysis. The result is that the program slices contain more security relevant statements, reducing the false-positive and false-negative rate of the neural network.

However, the previous studies treat source code as a sequence of tokens. This means important semantic information is lost, such as the program structure and control/data flow. Figure 14 shows an example of how SySeVR transforms a C program into tokens. There are two control flow paths through this program. The first control flow path allocates an array of size 10, sets the index variable to 9, and assigns a new value to the array on the specified index. If the condition on line 6 is true, an alternative flow will be executed. This flow sets the index to 19 before assigning a new value to the array.

In this example, the first control flow path is not vulnerable because index 9 is within range of the buffer's capacity. The alternative flow is vulnerable because 19 is out of range. However, this alternative flow is never executed because the if condition is always false and thus the program is safe.

In the SySeVR representation on the right, information about the control/data flow is not present. This could result in a wrong classification because the program can be interpreted in different ways. For example, it is unclear whether the execution of line 6 ($index = 19;$) depends on the if condition at line 4 ($if(false)$). If we assume line 6 is always executed, the code would be incorrectly labeled as vulnerable.

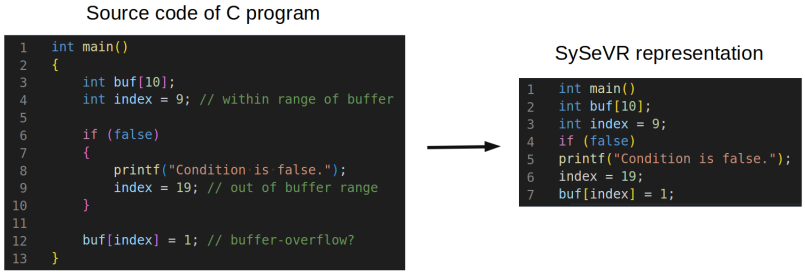

Figure 14: SySeVR representation of source code.

DEVIGN

To overcome these limitations, Zhou et al. [29] proposed Devign, a neural network that can operate on graph structures. This type of neural network is known as a graph neural network (GNN). GNN's are specifically designed to learn from graph data. Learning from this type of data is ideal for vulnerability detection because program syntax and semantics can be well expressed in graph structures. Examples of graphs that represent program syntax/semantics are abstract syntax trees, control flow graphs, and data flow graphs. By combining these representations, a neural network can reason about the security properties of a program by detecting patterns in its structure, control flow, and data flow. Their results show that representing a program as an abstract syntax tree (combined with a control/data flow graph) outperforms previous state-of-the-art approaches. These previous approaches include the BLSTM architecture used by SySeVR/VulDeePecker.

FUNDED

The limiting factor in Devign's work is that the graph neural network could not distinguish between control and data dependence edges. To overcome this issue, Wang et al. [22] propose FUNDED. This neural network uses *typed graphs* to distinguish between different edge types. In addition to using typed edges, they also included control dependence edges. Since this method captures more information in the graph structure, it can outperform alternative methods such as VulDeePecker and Devign.

3. RESEARCH DESIGN

This section describes the motivation behind the research, the goal, the research questions, and the research methods used.

3.1. RESEARCH MOTIVATION

Although deep-learning models have achieved promising results in vulnerability detection, there does not seem to be consensus among researchers on the appropriate level of granularity. Prior work distinguishes two levels of granularity: course grained and fine grained detection [29]. Coarse grained approaches predict if an entire file or program is vulnerable [6, 16]. Fine grained approaches focus on functions or program slices [12, 13, 22, 29].

Performing predictions at a coarse grained level makes it hard to pin down the exact location of a vulnerability because a file or program typically consists of many lines of code [12]. As a result, fine grained approaches are often used. However, to the best of our knowledge, no prior research has compared the function-level and program slicing approach.

This knowledge gap is interesting because the granularity level directly impacts what data is fed into the neural network. Based on this input data, the neural network makes a prediction. The prediction is only as good as the quality of the input data. If important information is missing, the neural network may misclassify a sample. Therefore, it is important to determine whether the function-level or program slicing approach captures the most security-relevant information.

We hypothesize that program slicing leads to better predictions than function-level analysis. The main reason is that program slicing captures the interaction between functions. Based on our experience, this information is often required for manual vulnerability detection. Analyzing a function in isolation is usually not enough to assess whether the code is vulnerable to exploitation. Often the parameters passed to the function must be analyzed to get a complete picture.

To compare the function-level approach with program slicing, we need an existing deep-learning model to run experiments on. We chose the deep-learning model designed by Wang et al. [22] called FUNDED. Their reported results show that FUNDED outperforms prior work at the function-level. The distinguishing factor in their work is the use of neural networks that operate on graphs. In a graph structure, the syntax and semantics of a program can be well expressed, resulting in a more accurate model.

Despite these promising results, there is still room for improvement. The authors report a F1 score of 92%. Although this is a big step forward, in practice this still results in false-positives and false-negatives. False-positives lead a security expert to waste time researching vulnerabilities that do not exist. False-negatives cause a security expert to miss certain vulnerabilities.

Both of these errors prevent the efficient detection of software vulnerabilities. By extending FUNDED with program slicing, we expect these errors will decrease. Efficient vulnerability detection is important to create secure software that can withstand attacks. Repelling attacks is key to protect society against cyber attacks such as data theft or disruptions.

3.2. Research goal

This project aims to improve previous approaches by creating a GNN that detects vulnerabilities across a series of functions. This is based on the hypothesis that vulnerabilities can occur across function boundaries. To achieve this goal, the model FUNDED [22] is extended using a technique called *program slicing*.

Using program slicing for vulnerability detection is not new, but we are one of the first studies to combine slicing with graph neural networks. In addition to the main goal, we also improve the word embedding strategy used by FUNDED and include more language constructs in the program slicing algorithm compared to previous studies.

3.3. Research questions

Based on the hypothesis that vulnerabilities occur across function boundaries, the following main research question is formulated:

- How can incorporating program slicing improve a graph neural network's ability to detect buffer vulnerabilities?

To answer the main question, the following subquestions need to be addressed. Each question has a sequential number and is prefixed by RQ (research question).

RQ1 How can program slicing be applied to preserve behavior related to buffer vulnerabilities?

RQ2 How does the target depth of the program slicing algorithm affect the model's overall accuracy?

RQ3 How does the GLICE model compare to the original FUNDED model in training duration and accuracy?

RQ1: How can program slicing be applied to preserve behavior related to buffer vulnerabilities?

This question tries to apply Mark Weiser's program slicing algorithm to preserve program behavior related to buffer vulnerabilities. We limit ourselves to buffer vulnerabilities because this allows for more targeted research into this common vulnerability type [44]. The output of this algorithm should be a subprogram (also known as slice) that contains program statements relevant to a vulnerability.

This research question builds on prior studies that have applied program slicing for vulnerability detection, namely VulDeePecker and SySeVR [12, 13]. These studies claim to extract slices that contain information relevant to a vulnerability's existence. However, it is unclear to what extent these algorithms preserve security-related statements. For example, they did not measure the percentage of relevant statements extracted. Our goal is to perform these measurements and improve the slicing algorithm in case of missing information. The results should be a set of program slices that contain all the security-related information from our data set. Using a slice, a security expert or deep-learning model should be able to assess whether it is vulnerable.

Q2: How does the target depth of the program slicing algorithm affect the model's overall accuracy?

As mentioned earlier, program slicing analyzes multiple functions in case a vulnerability arises due to the interaction between functions. Functions interact with each other by function calls. A function call passes control and (potentially) arguments to another function or itself (recursive call). A caller is a function that calls another function and the callee is the function being called.

A caller can both call another function but also be called itself. A call tree represents the calling relationship between functions. An edge in this directed graph represents a function call and the nodes represent functions. If we perform program slicing on the entire call tree, the resulting program slice may become large. A graph that is too large may prevent the model (graph neural network) from efficiently extracting patterns. To test this hypothesis, we analyze the performance of the model at different target depths.

We define target depth as the upper limit allowed for a function's depth in the call tree. The depth is equal to the length of the shortest path to the root node. The root node is the point of interest (also known as slicing criterion) for the program slicing algorithm. It is also the highest node in the tree structure and has no parent node. As we increase the target depth allowed in the call graph, the number of functions included in the slice will also increase (assuming there are callers).

To illustrate this point, let there be a program A. We are interested in preserving behavior related to a (potential) vulnerability in function A. Function A is called by function B and function B is called by function C. The largest number of edges we have to traverse is from function C to function A. The length of this path is two because we have to traverse two edges. This means that decreasing the target depth below two excludes functions from the slice. Figure 15 visualizes this example, the dashed circles indicate the function call depth.

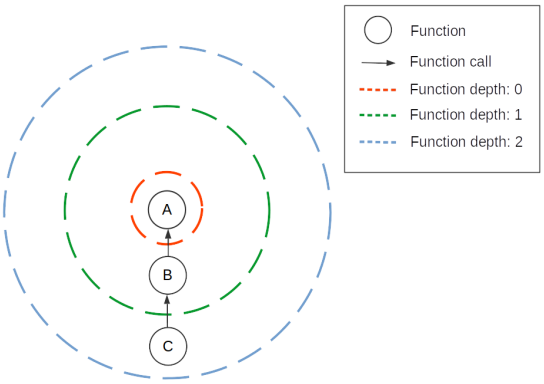

Figure 15: Example of a call tree.

We hypothesize that increasing the target depth will improve the overall accuracy of the model. However, increasing the function depth beyond a certain point may reduce the model's ability to efficiently extract patterns. To measure this, we will evaluate our model on a data set of buffer vulnerabilities.

Q3: How does the GLICE model compare to the original FUNDED model in training duration and accuracy?

Based on the results of the previous question, we choose the best-performing target depth for the model. This new model is called GLICE which stands for (g)raph neural network and program s(lice). GLICE is compared against FUNDED which uses the function-level approach. We compare both the training duration and accuracy because there may be a trade-off between those two. The training duration is defined as the time required to complete one epoch and the number of epochs until reaching the F1 score of FUNDED. After evaluating the results, we can answer the main question: how can program slicing improve a graph neural network's ability to detect buffer vulnerabilities?

3.4. Research method

To answer the above-mentioned questions, empirical research is conducted. The impact of incorporating program slicing is evaluated by collecting a data set of (non-)vulnerable code. By creating a modified version of the FUNDED model that uses program slicing, certain performance metrics can be compared against the original function-level approach.

RQ1: How can program slicing be applied to preserve behavior related to buffer vulnerabilities?

A literature review is performed to address the first subquestion about how to apply program slicing. To find a slicing algorithm that is suitable for vulnerability detection, we review the existing literature. The program slicing algorithms of VulDeePecker and SySeVR [12, 13] are important to review because they have similar requirements. The shared requirements are that the algorithm must preserve behavior related to (buffer) vulnerabilities and support inter-procedural analysis. Another shared requirement is that it supports the statically typed languages C and C++. A key difference between our program slicing algorithm and VulDeePecker/SySeVR is that we preserve the graph structure instead of converting it to a token sequence.

To evaluate and improve VulDeePecker/SySeVR, we validate that lines marked as vulnerable are included in the resulting slice. We perform this experiment on the samples from SARD because this data set labeled vulnerable lines. In case not all vulnerable lines are included, changes will be made to the algorithm.

Q2: How does the target depth of the program slicing algorithm affect the model's overall accuracy?

After answering the previous question, the FUNDED model is reproduced. To validate that our reproduced model is working as expected, we first compare the model against the results reported by the original authors. To perform this comparison, a subset of the data from the FUNDED paper is used. We evaluate our model against this subset and compare the accuracy against the reported results.

Next, the FUNDED model is modified to use the program slicing algorithm. The result is a new model called GLICE that performs inter-procedural program slicing. To answer this subquestion, experiments are conducted on the GLICE model to determine the optimal target depth of the slicing algorithm.

A higher target depth increases the size of the graph (assuming there are calling functions). Although a larger graph contains more security-related statements, it is questionable whether the model can efficiently extract patterns from this larger graph. To determine the optimal target depth, experiments are conducted with different target depths. These experiments are conducted on a data set of buffer vulnerabilities because these vulnerabilities are common and critical according to MITRE. The performance of the model is measured using the metrics precision, recall, F1 score, and accuracy.

Precision is the percentage of identified vulnerabilities that are correctly categorized. Recall is the percentage of vulnerabilities discovered when looking at the total number of vulnerabilities present. F1 is the harmonic mean of the precision and recall. Finally, accuracy is the number of correctly classified samples divided by the total number of samples.

Table 1 shows the corresponding formulas per metric. A false positive (FP) incorrectly indicates the presence of a vulnerability. A false negative (FN) indicates the opposite error, it states the absence of a vulnerability when it is actually present. True positives (TP) or true negatives (TN) indicate a correctly classified vulnerable or non-vulnerable sample, respectively.

Metric	Formula
Precision	$\frac{TP}{TP+FP}$
Recall	$\frac{TP}{TP+FN}$
F1 score	$2 \times \frac{Precision \times Recall}{Precision+Recall}$
Accuracy	$\frac{TP+TN}{TP+TN+FP+FN}$

Table 1: Formulas per metric.

To guarantee a fair comparison of different target depths, we used the same hyperparameters for GLICE and FUNDED. This ensures that both models have the same number of configurable weights and thus the same expressive power. The chosen hyperparameters were taken from the original FUNDED paper. The authors of FUNDED tuned these hyperparameters by performing a grid search. A grid search performs an extensive search through a manually defined subset of hyperparameters. The hyperparameters include variables such as the optimizer and learning rate used.

Finally, to ensure reproducibility of the experiment we used the k-fold cross validation procedure. This resampling procedure evaluates the skill of a model on unseen data. Instead of evaluating our model once, we evaluate it k times. A value of 10 will be used for k because it has been found to have a lower bias [11]. The procedure is shown below:

1. Split the data set randomly into k groups;
2. For each group:
   (a) Use the group as the test data set;
   (b) Use the remaining groups for training/validation;
   (c) Train the model on the training set;
   (d) Evaluate the model on the test data set and store the results.
3. Take the average of the results.

Q3: How does the GLICE model compare to the original FUNDED model in training duration and accuracy?

The best-performing GLICE model from the previous subquestion is compared against the original FUNDED model to evaluate whether program slicing positively affects the model's performance. The performance is determined by comparing the precision, recall, F1 score, and accuracy. In addition, the training duration is compared by plotting the accuracy of the models against training time. The same hyperparameters are used for both models to ensure that the underlying graph neural networks have the same expressive power.

Figure 16 shows a summary of the research performed for research question two and three. The color indicates the preprocessing used by the model which can be either function-level preprocessing or program slicing. The preprocessing, splitting, and training step is performed on the buffer vulnerability data set.

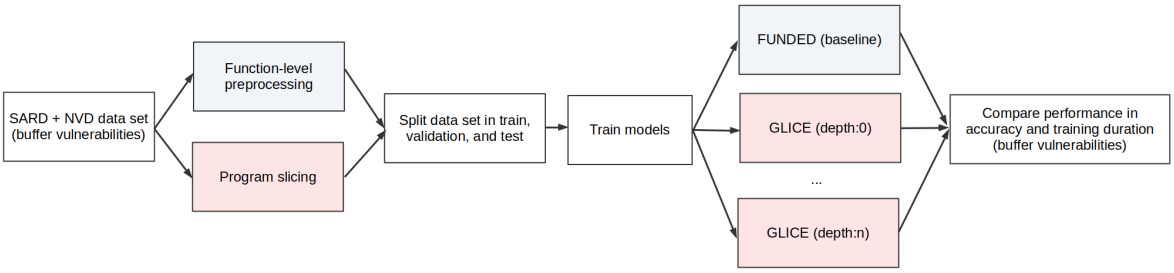

Figure 16: Summary of research performed.

4. IMPLEMENTATION

This section describes the data set, model, and algorithm used to conduct the experiments. The first section describes how the data set is collected and labeled. The second section describes how the deep-learning model is reproduced. Finally, the program slicing algorithm is defined addressing the first research question.

4.1. DATA SET

At the time of writing, the data set of FUNDED is only partially available. That is why we used two other sources of vulnerable and non-vulnerable code samples, the National Vulnerability Database (NVD) [45] and Software Assurance Reference Dataset (SARD) [46]. Both of these databases are maintained by the National Institute of Standards and Technology (NIST). The NVD contains information about publicly known vulnerabilities. This enables security analysts to get an overview of real-world vulnerabilities and assess whether their systems are affected.

The purpose of SARD is different. It does not give an up-to-date overview of real-world vulnerabilities, but it provides (non-)vulnerable test cases. Some of these test cases are taken from real software applications, but the majority are synthetic. This means that they are manually written by security experts or generated using templates.

Li et al. [13] have already combined these two sources in a publicly available data set. We use their data set to speed up the data collection process. Especially collecting samples from the NVD is time-consuming because a reference to the commit (that fixes a vulnerability) is often missing. This prevents the automated extraction of vulnerable and non-vulnerable samples from a repository.

A subset of the data set from Li et al. [13] is used because our research is limited to buffer vulnerabilities. After removing unrelated vulnerability types, the data set contains more than 25,000 C/C++ samples. This includes the CWE's listed below. No samples were found for CWE-123 (Write-what-where), so we excluded this vulnerability type from our analysis.

- CWE-787: Out-of-bounds Write
  - CWE-121: Stack-based Buffer Overflow
  - CWE-122: Heap-based Buffer Overflow
  - CWE-124: Buffer Underwrite
- CWE-125: Out-of-bounds Read
  - CWE-126: Buffer Over-read
  - CWE-127: Buffer Under-read

In Table 2 an overview of the number of samples per category is given. The two largest categories are the stack-based and heap-based buffer overflow samples from SARD. These two categories account for 63% of the data set. If we compare the number of vulnerable and non-vulnerable samples, we notice that the non-vulnerable group is larger.

The non-vulnerable samples account for 61% of the total number of samples. This means that our data set is imbalanced. An imbalanced data set may result in a classifier reaching circa 61% accuracy by always guessing non-vulnerable. In order to solve this problem we randomly removed 5,622 non-vulnerable samples from the data set. The result is a data set that contains both 10,123 vulnerable and non-vulnerable samples.

Source	Category	Number of samples	Percentage vulnerable
SARD	Stack-based Buffer Overflow	10,015	40%
SARD	Heap-based Buffer Overflow	6,377	39%
SARD	Buffer Underwrite	3,382	38%
SARD	Buffer Over-read	2,664	36%
SARD	Buffer Under-read	3,382	38%
NVD	Buffer Overflow/Over-read	48	72%
Total		25,868	39%

Table 2: Number of samples per category.

Next, we analyze at which line the vulnerability occurs. This information is useful for the slicing criterion used by our program slicing algorithm. The vulnerable lines are already labeled for the SARD data set and we manually label them for the NVD data set. Based on this analysis, we discover two categories. The first category of vulnerabilities are caused by the array access operator. For example, if a buffer is 10 characters long (*char buffer[10];*) and a program writes to index 20 then a buffer overflow occurs (*buffer[20] = 'A';*). The second category is caused by the misuse of certain library functions. For example, the function "strcpy" which copies a C string to a new location can overflow. This function may overflow if the C string passed as the source is longer than the size of the destination buffer. A complete table of functions that are misused is shown in Table 3.

Function signature	Description
void* **memset**(void* dest, int c, size_t num)	Set a number of bytes (num) of the destination buffer (dest) to the specified value (c).
void* **memmove**(void* dest, void* source, size_t num)	Copies a number of bytes (num) of the source buffer to the destination (dest).
void* **memcpy**(void* dest, void* source, size_t num)	Copies a number of bytes (num) of the source buffer to the destination (dest). The main difference with memmove is that the buffers are not allowed to overlap.
char* **strcpy**(char* dest, char* source)	Copies the source string to the destination (dest).
char* **strncpy**(char* dest, char* source, size_t num)	Copies a number of characters (num) of the source string to the destination (dest).
wchar_t* **wcscpy**(wchar_t* dest, wchar_t* source)	Similar to strcpy, but for wide characters.
wchar_t* **wcsncpy**(wchar_t* dest, wchar_t* source, size_t num)	Similar to strncpy, but for wide characters.
char* **strcat**(char* dest, char* source)	Append the source string to the destination (dest) string.
char* **strncat**(char* dest, char* source, size_t num)	Append a number of characters (num) from the source string to the destination (dest) string.
wchar_t* **wcscat**(wchar_t* dest, wchar_t* source)	Similar to strcat, but for wide characters.
wchar_t* **wcsncat**(wchar_t* dest, wchar_t* source, size_t num)	Similar to strncat, but for wide characters.
ssize_t **recv**(int s, void *buf, size_t len, int flags)	Receive data from socket (s) and write it to buffer (buf) with size (len).
char* **gets**(char* str)	Receive data from standard input and write it to the string (str).
char* **fgets**(char* str, int num, FILE* stream)	Receive a number of characters (num) from the stream and write it to the string (str).
int **scanf**(char* format, ...)	Read data from the standard input and write it to the additional parameters according to the format.
int **fscanf**(FILE* stream, char* format, ...)	Similar to scanf, but reads data from stream.

Table 3: C/C++ library function calls that can cause buffer vulnerabilities. The table is based on the current C17 standard[32].

LABELING

The labeling strategy used by the authors of FUNDED is based on the method name. A method name containing the word "bad" is marked vulnerable and a method name containing the word "good" is marked non-vulnerable. This strategy seems reasonable because the SARD test cases use this predictable naming convention. To validate if the labeling strategy works correctly, we selected a subset of 100 samples from the data set of FUNDED. We used the FUNDED data set instead of our own because we want to highlight the shortcomings in their research. Moreover, their labeling strategy was not designed for our buffer vulnerability data set and thus incorrect labels would be expected.

Based on manual analysis, we determined that 19% of the samples were correctly labeled (see Table 4). For the other samples, we could not determine whether the code was vulnerable. The root cause was that the source function, sink function or both were missing. The source function is where data comes from and the sink function is where data ends up. The potential vulnerability occurs in the sink function. If we would assume that these missing source/sink functions are safe, 35% of the samples would have the wrong label. On the other hand, if we would assume the source/sink functions are vulnerable, 46% of the samples would have the wrong label.

	Percentage of samples
Correct label (good)	10%
Correct label (bad)	9%
Source function missing (good)	13%
Source function missing (bad)	13%
Sink function missing (good)	8%
Sink function missing (bad)	14%
Source and sink function missing (good)	25%
Source and sink function missing (bad)	8%

Table 4: Number of samples (in)correctly labeled.

The labeling strategy used by SySeVR does not have the same limitation. Their model analyzes both the source and sink function using program slicing. A slice from SARD is labeled as vulnerable if it contains a vulnerable line. All vulnerable lines are listed in a XML document included in the data set. The labeling strategy used for slices from NVD is more complex. A line is marked vulnerable if it is removed in the commit patching the vulnerability. We will use a similar labeling strategy for our data set.

However, SySeVR has a different problem: duplication. Their program slicing algorithm may define multiple slicing criteria for a single function. Each slicing criterion generates a program slice. Multiple criteria per function results in several program slices that can overlap. If slices overlap, there is duplication in our data set. This will produce overly optimistic results because samples in the test set may be similar to those in the train set. To prevent this problem, we will only allow a single slicing criterion per function.

4.2. MODEL

FUNDED is a deep-learning model designed to detect vulnerabilities. The input of the model is a function. Based on the source code of this function, it constructs a directed graph. This graph is an abstract syntax tree (AST) enriched with control flow (CFG), data flow (DFG), and data dependence (DDG) information. The nodes in the graph are statements and expressions. The edges represent different relationships, such as parent-child (in the abstract syntax tree) or control flow. The nodes in the graph are converted to vectors using a technique called Word2Vec.

LIBRARIES

The source code is publicly available at Github [49]. The model is implemented in Python 3.7 [33] and the graph neural network is built using Tensorflow 2.0 [34]. Tensorflow is a popular Python library for machine learning. The authors of FUNDED did not implement the graph neural network themselves but used an implementation provided by Microsoft [30]. This implementation provides multiple graph neural network architectures including GGNN (Gated Graph Neural Networks) and RGCN (Relational Graph Convolutional Networks). The authors chose the GGNN architecture because the RGCN architecture required substantially more parameters for larger graphs. This increase in parameters can cause overfitting, especially when there is a small number of training samples available [26]. Table 5 below shows an overview of all Python libraries required to run the model:

Python library	Description
Tensorflow	Machine learning library used to implement the graph neural network.
NumPy	Library used for working with arrays.
DPU_utils	Set of utilities for deep learning.
Gensim	Implements the Word2Vec model. This model is used to convert the tokens to vectors.
Sklearn	Splits the data set and calculates the precision, recall, F1 score, and accuracy.
NNI	NNI stands for Neural Network Intelligence. It optimizes the hyperparameters of the neural network using grid search.

Table 5: Python libraries used by the deep-learning model.

CUDA

The first step to run the model is to install the CUDA toolkit. This toolkit provides an API for Tensorflow to perform computations on the GPU. We installed version 11.2 which supports our NVIDIA RTX 3060 GPU and includes several performance improvements. The problem was that Tensorflow version 2.0 only support CUDA 10.0. This CUDA version is incompatible with our GPU. To fix this issue, Tensorflow was updated to version: 2.9.0. This version of Tensorflow targets CUDA 11.2.

ARCHITECTURE

FUNDED consists of two subsystems: a preprocessing system and the deep-learning model. The preprocessing system is responsible for converting source code into labeled graphs. These graphs represent the abstract syntax tree enhanced with control flow, data flow, and control dependence information. The deep-learning model trains and evaluates the neural network based on these labeled graphs.

Generating graphs from source code is a complex task. To perform this preprocessing task, FUNDED uses two external libraries: CDT (v5.6.0) and Joern (v1.0.92). CDT stands for C/C++ Development Tooling and is developed by the Eclipse Foundation. This library is used in the Eclipse IDE (Integrated Development Environment) for syntax highlighting and source code refactoring/generation. FUNDED uses their library for parsing the source code and generating an abstract syntax tree.

Joern is a code analysis platform that support C/C++ applications. This code analysis platform is used by FUNDED to generate code property graphs [25]. A code property graph consists of nodes and their types (e.g. methods, variables, and control structures) and labeled edges. These edges represent control flow, control dependence, data flow and parent-child relations (AST). We found it unclear why two libraries are used to generate graphs, as CDT does not contain extra edges that are not already present in Joern. The list below shows how Joern and CDT are combined to generate the final graph representation:

1. Extract all .c files from project;
2. Extract functions from .c files;
3. Parse functions from files (using Eclipse CDT):
   (a) Generate graph: AST;
   (b) Label functions as vulnerable/non-vulnerable;
   (c) Export graph to file.
4. Parse function from files (using Joern):
   (a) Generate graphs: AST, CFG, DFG, DDG;
   (b) Export graphs to file.
5. Combine graphs generated by Eclipse CDT with Joern.

The deep-learning model consists of six components. Command line input enters the application through the CLI component. This component calls CLI Utility to start the training procedure. CLI Utility uses the Data component to load the graphs from disk into memory. After which, the Model component is used to build the neural network. This neural network consists of layers which are defined in the Layer module. Activation functions used in these layers are defined in the Utility module.

Component	Description
CLI	Parses command line input.
CLI Utility	Implements the training and evaluation procedure.
Data	Loads data set from disk into memory.
Utility	Implements utilities such as activation functions.
Layer	Implements the layers of the neural network.
Model	Combines the layers to build the graph neural network.

Table 6: Components of the deep-learning model.

The steps below show how the training and evaluation procedure works:

1. Convert program tokens to vectors using Word2Vec model;
2. Split data set in train, validation, and test;
3. Load data set and graph neural network;
4. Run epochs:
   (a) Shuffle training samples;
   (b) Split training samples in batches;
   (c) For each batch:
      i. Perform predictions on training samples;
      ii. Calculate loss on training samples;
      iii. Update weights based on loss.
   (d) Evaluate on validation set.
5. Evaluate on test set.

TRAIN/VALIDATION/TEST SPLIT

The data set is split in a train, validation and test set. The training set is used by the model to learn patterns in the data. These patterns are captured in the weights of the neural network. The validation set is used to evaluate the model performance during training.

As soon as the model stops improving for the validation set, training is stopped. This technique is called early stopping and prevents the neural network from overfitting. Another advantage is that the optimal number of training epochs is automatically determined. The downside of early stopping is that the training process can be noisy and the performance metrics may go down before going up. This means that the first indication of overfitting may not be the best place to stop. That is why the authors introduced patience. This is a delay in epochs before the model stops training. In other words, the model has to stop improving for a certain number of epochs.

The test set is used to evaluate the final performance of the model. Since the number of epochs was optimized for the validation set, the validation results become biased. The number of epochs is not optimized for the test set and thus provides unbiased results.

The data set was split in the ratio 80% (train), 10% (validation) and 10% (test). Currently, there is no consensus in the machine learning community on the best ratios but these percentages are commonly used. The k-fold cross-validation procedure is used to provide less biased results.

CRITICAL ISSUES

By running the model and analyzing the source code, the following critical issues were discovered:

1. Out-of-memory exceptions occurred because the entire data set was loaded into memory. This exceeded the maximum amount of memory in the desktop PC which was 32GB.

2. There are logical errors in the code that parses the graphs. One-based numbering is used for the CDT nodes. Zero-based numbering is used for the Joern nodes. The problem is that the model assumes zero-based numbering. This causes multiple errors while parsing the CDT graphs including incorrectly connecting edges and removing (a small number of) edges.

3. The graphs from CDT and Joern are incorrectly combined. First both graphs are shuffled and after shuffling they are combined. The labels from the CDT graphs are used, causing wrongly labeled Joern graphs.

4. Training occurs on both samples from the training set and validation set. The validation set should only be used to evaluate the model and optimize the hyperparameters, such as the number of epochs. If the model is also trained on the validation set, the neural network may not choose the best hyperparameters for unseen data. This may cause the model to overfit on the training and validation set and thus degrade the final metrics calculated for the test set.

5. The precision, recall, F1 score, and accuracy are calculated over a subset of the test set. Calculating these metrics over a subset, results in incorrect assumptions about the model's performance. This error occurs because these metrics are calculated over the last batch of the test set instead of over all test samples.

The first issue was resolved by streaming batches of data from disk. To prevent waiting on I/O operations, the next batch is loaded while training. Caching was used to ensure that data conversions to Numpy arrays are performed once. To implement caching, we used Pickle. This is a module in the standard Python Library which can serialize and deserialize Python objects. During serialization an object is converted into a byte stream and during deserialization the byte stream is converted back into the original object.

The second issue was resolved by converting CDT nodes to zero-based numbering. The third issue was resolved by combining the CDT and Joern graphs before shuffling instead of after shuffling. Afterward, the fourth issue was resolved by removing the Python code that adds samples from the validation set to the training set. Finally, the last issue was resolved by combining the batches before calculating the final performance metrics.

REPORTED RESULTS

The number of critical issues discovered in the FUNDED model raises questions on the validity of the results reported by the original authors. Issues related to parsing and combining the CDT and Joern graphs could negatively affect the performance of the model. Furthermore, errors in calculating the performance metrics can undermine the conclusions drawn by the authors.

To validate the conclusions drawn by the authors, we reproduce the results for a subset of their data. We can not use the entire data set because only a subset is publicly available. This subset contains OS command injection vulnerabilities (CWE-78) and is used train and evaluate our reproduced model. Figure 17 shows the results of our experiment compared to the results reported by the original authors. Based on the results, we conclude that our reproduced model has a similar level of performance for CWE-78. The precision may be better and the recall slightly worse but the F1 score is the same. Furthermore, the 1% increase in accuracy can be explained by the fixes for issue two and three.

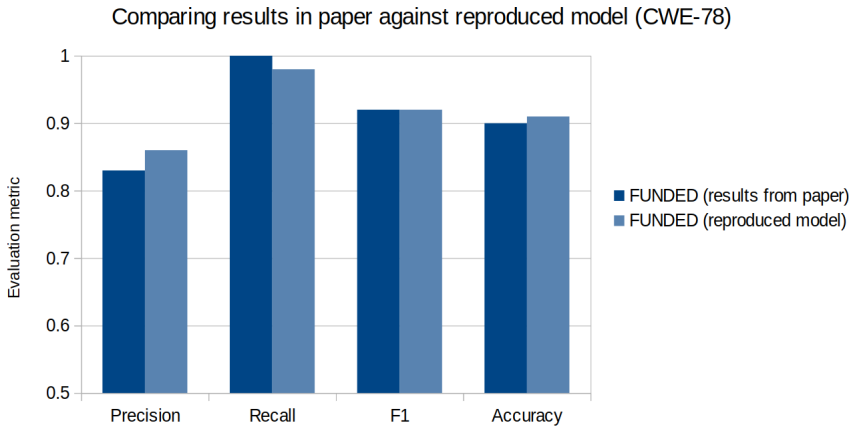

Figure 17: Results in paper versus reproduced model (CWE-78: OS Command Injection).

WORD2VEC

While analyzing the source code for bugs, we also noticed several improvements that could be made to the Word2Vec model. FUNDED uses Word2Vec to convert tokens to vectors. A token is the smallest unit of a program that is significant to the compiler. Tokens include keywords (int, float, if, else), variable names (data, buffer), and comparison operators (<, >, ==). A vector is simply a list of numbers. Tokens need to be converted into this numerical form because an artificial neural network cannot directly operate on text.

Word2Vec was originally designed for natural language processing (NLP). NLP is the ability for computers to understand human languages such as English. Its purpose was to generate vectors in such a way that similar words are close together in vector space. The method is supervised but can learn from unlabeled inputs. Word2Vec learns by looking at the context of a target word to determine its meaning. The context are the surrounding words of the target word in a sentence. It is based on the assumption that similar words appear in the same places in other sentences.

Word2Vec has been used before to generate embeddings for tokens. The main difference is that the model is trained on tokens from a program instead of words from a natural language such as English. Another key difference is that there are no sentences in a program. Instead of a sentence, a line of code, function or file can be used.

The following limitations are present in the word embeddings strategy used by FUNDED:

1. A large portion of the sentences contain only a single token. This is a problem because Word2Vec bases the vector representation on surrounding tokens. If there are no surrounding tokens, the model is unable to choose a good vector representation based on its context.

2. Splitting a line into tokens often incurs errors. For example, the line "printWLine(data);" is seen as a single token. This line should be split into two tokens: the method call "printWLine" and the argument "data". The root cause of this bug is that a line of code is split by the space character. The problem is that this bug results in more unique tokens which lowers the frequency of each token. If tokens have a lower frequency, the Word2Vec model has less examples of how the token is used in different contexts. Less examples may result in lower quality vector representations.

3. Tokens after the closing parenthesis ")" are removed because of a bug in the parser. The parser removes information after the closing parenthesis because this symbol is used to signify an end of line. The problem is that this symbol is also commonly used for method calls or control structures (e.g. "strcpy(...)", "if(...)", "for(...)", and "while(...)"). This results in the method call "strcpy(func1(), str2)" to become "strcpy(func1(".

4. Variable names are not renamed to var_0, var_1, var_2 etc. The main problem with using the original variable names is that the SARD data set contains variables such as "dataGoodBuffer" and "dataBadBuffer". These variable names indicate whether the buffer is large enough to prevent a buffer overflow. In real-world software, variables do not have such an obvious name and thus the model may learn patterns that are not applicable to samples outside the SARD data set.

The first issue was resolved by changing the definition of a sentence. In the FUNDED implementation a sentence was either a line of code or a single token. In our new embedding strategy, a sentence represents a function. This ensures there are always surrounding tokens from the same line or other closely located lines.

The second and third issue resolved around incorrectly splitting a line of code in tokens. By using the tokenization logic from Joern, both issues were resolved. Lastly, we implement variable renaming to prevent the model from finding patterns in the variable names. Figure 18 shows an example of how the resulting graph looks.

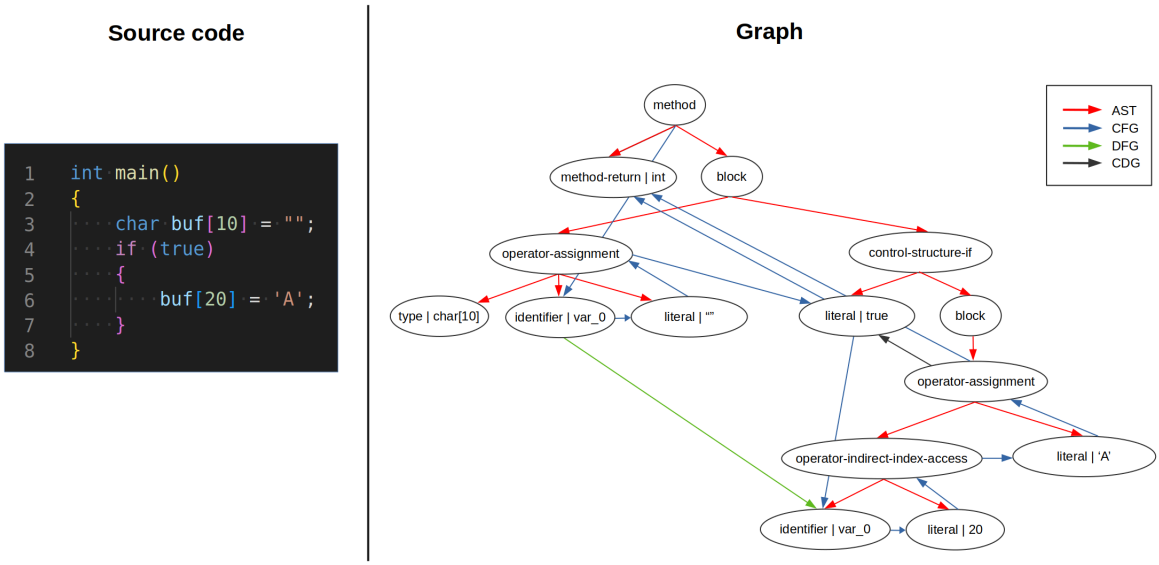

Figure 18: Example graph of the function main. Bug in tokenization logic is fixed and variables are renamed.

Figure 19 compares the performance of the original FUNDED model against our new model (called GLICE). This comparison is based on the buffer vulnerability data set. The result show a statistically significant increase of 1.4% in F1 score. This improvement can be explained by the new embedding model which improves precision.

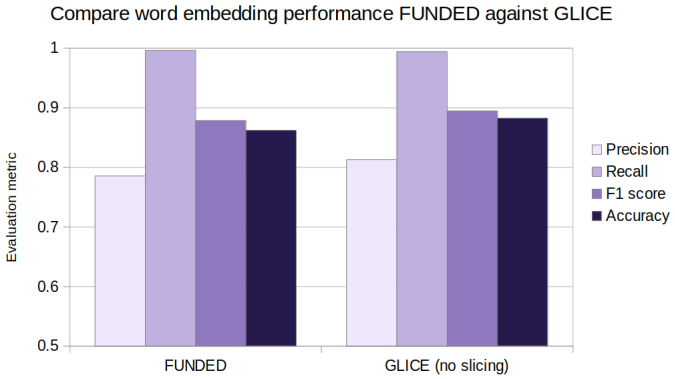

Figure 19: Compare word embedding performance FUNDED vs GLICE.

4.3. PROGRAM SLICING

As mentioned earlier, our program slicing algorithm extends the work of Mark Weiser. Inspired by VulDeePecker/SySeVR, we use security-related library functions and array access operators as slicing criterion. The differentiating factor in our work is that we focus on graph neural networks. These networks require a graph as input which is why we need to preserve the edges. The control and data dependence edges can be retrieved by storing all traversed edges. Control flow edges and parent-child edges (AST) need to be recomputed.

In addition to preserving the graph structure, we made several key contributions to VulDeePecker and SySeVR. These contributions were based on manual analysis of the slices. During this analysis, we noticed that crucial information was missing to determine whether the code was vulnerable. This missing information was related to the following language constructs:

- Function pointers;
- Global variables;
- Macros;
- Structures;

The first limitation is related to the inability to resolve function pointers. A function pointer stores the address of a function so that it can be called later. In other words, it points to code instead of data. The inability to resolve a function pointer makes it impossible to determine which code is executed when it is called. In our data set, 4% of the samples require function pointer analysis to determine if the code is vulnerable.

Another limitation is that the algorithm does not include global variables in the program slice. A global variable is declared outside the scope of a function. This type of variable can be accessed and modified by any function. In case the variable is marked as constant, modifications are not allowed. In 17% of the samples, global variables are defined. The values assigned to these variables are not included in the program slice. This is a severe limitation because the presence of a vulnerability often depends on these values.

The third limitation is that the algorithm is unable to resolve macros. A macro is a piece of code that can be given a name. When the name is used, it is replaced by the code defined in the macro. In the VulDeePecker/SySeVR implementation only the name of the macro is visible and not the piece of code. As a result, it is not possible to determine what code is executed at run time.

The last limitation is related to structures. A structure (also known as struct) can be used to define custom data types. This user defined type holds a group of related variables in one place. Each variable in the structure is known as a member. Each member may be of a different type (e.g. int, float, double). In the VulDeePecker/SySeVR implementation these structures are not included in the program slice. This is a limitation because information about the type of a member can be security-relevant. For example, the size of an array may determine whether a buffer overflow can occur.

5. EXPERIMENTAL RESULTS

This section describes the results of our experiments. The first subsection answers research question one which evaluates our program slicing algorithm. Next, research question two is answered about the optimal target depth of our algorithm. The optimal depth of our slicing algorithm is the depth that yields the highest F1 score. Finally, we answer research question three by comparing the best-performing GLICE model of the previous question against FUNDED.

5.1. SLICING ALGORITHM

RQ1 raises the question: how can program slicing be applied to preserve behavior related to buffer vulnerabilities? To determine if our slicing implementation addresses this question, we experiment on the buffer vulnerability data set. By validating that lines marked as vulnerable are included in the resulting slice, we can be certain that behavior related to buffer vulnerabilities is preserved.

We performed this experiment on the samples from SARD because they labeled vulnerable lines with comments such as "FIX: buffer too small" or "POTENTIAL FLAW: buffer overflow". VulDeePecker/SySeVR's version of the algorithm included 94% of all vulnerable lines. By fixing the limitations described in subsection 4.3, 100% of the vulnerable lines where included.

5.2. OPTIMAL TARGET DEPTH

This section addresses research question two: how does the target depth of the program slicing algorithm affect the model's overall accuracy? We evaluate this by increasing the target depth from 0 to 4 and measuring the accuracy at each step. A target depth of 0, means we analyze only the function where the vulnerability occurs. If the target depth is larger than 0, we iteratively analyze the callers of a function. The analysis is performed until the depth is equal to 4, because 4 is the maximum depth in our data set. The optimal target depth of our program slicing algorithm is the depth that yields the highest F1 score.

Figure 20 shows the results of our experiment. The F1 score of the model increases as we increase the target depth. The largest increase in performance is after increasing the function depth to 1. The F1 score increases by 8.4% compared to function depth 0. Thereafter, the model increases at a lower rate (0.6%, 1.0%, 1.0%).

Precision and recall are a tradeoff. Typically to increase precision for a given model implies lowering recall. Figure 21 shows the precision versus recall as the function depth increases. A function depth of 4 has both the highest precision and recall. The increase in precision has the highest impact on the rising F1 score. The recall improves only 0.4% and the precision improves 18.6% between function depth 0 and 4.

After increasing the functions depth from 0 to 1, the recall drops by 0.2. It seems the model increases precision at the cost of the recall score. This is supported by the 13.8% increase in precision between these two depths.

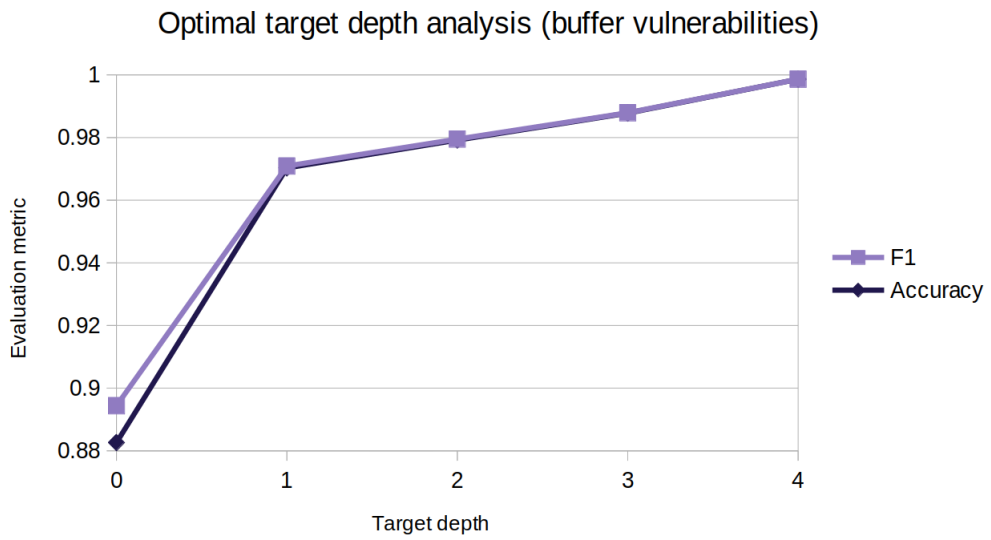

Figure 20: Analysis of the optimal target depth for our program slicing algorithm (F1 score versus accuracy).

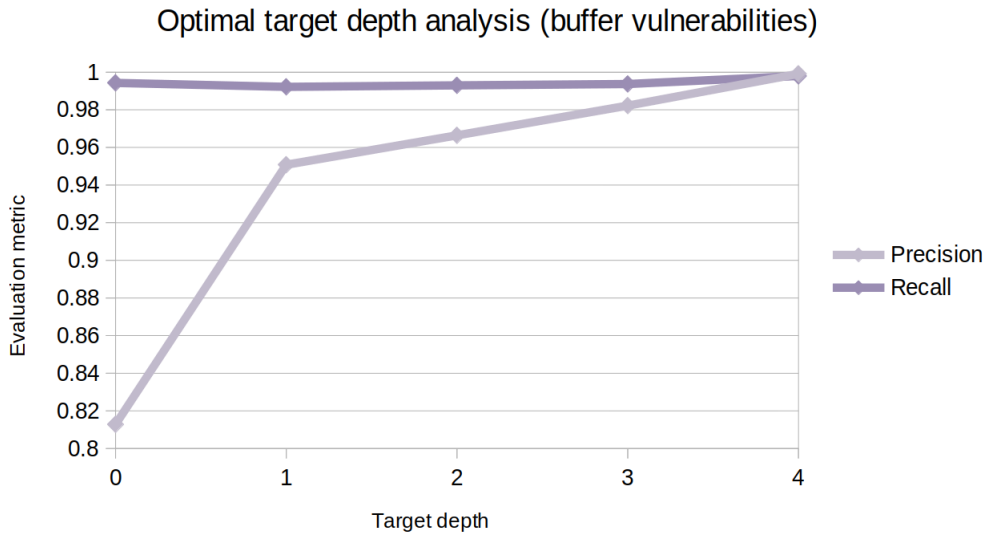

Figure 21: Analysis of the optimal target depth for our program slicing algorithm. (precision vs recall).

To investigate why the F1 score increases slower after changing the depth from 0 to 1, we split the data set in five groups (see Table 7). The largest group 64.81% are samples where the source and sink are defined in a single function. Inter-procedural slicing does not add additional calling context to these samples, because there are no callers. Samples with a maximum depth of 1 or larger do benefit from inter procedural slicing. These samples require deeper analysis because the source and sink are defined in different functions.

Sample's maximum depth	Number of samples	Percentage
0	16734	64.81%
1	7358	28.50%
2	576	2.23%
3	576	2.23%
4	576	2.23%

Table 7: Data set split by the maximum depth. A sample reaches the maximum depth when both the source and sink are included in the slice.

It is important to note that the samples that benefit from a depth equal or larger than 2 is small. This subset of the data is only 6.69% of the total. This also explains why increasing the target depth beyond 1, only increases the F1 score by 2.8%.

Figure 22 shows which group causes the increase in F1 score. As expected, samples with a maximum depth of zero do not improve when increasing the target depth. In these samples, the source and sink are defined in the same function. Samples with a maximum depth of 1 cause the largest increase in F1 score when changing the target depth from 0 to 1. This can be explained by the fact that these samples make up a large part of the data set: 28.50%. Table 8 shows the underlying table because an increase of 1% is hard to read from the chart.

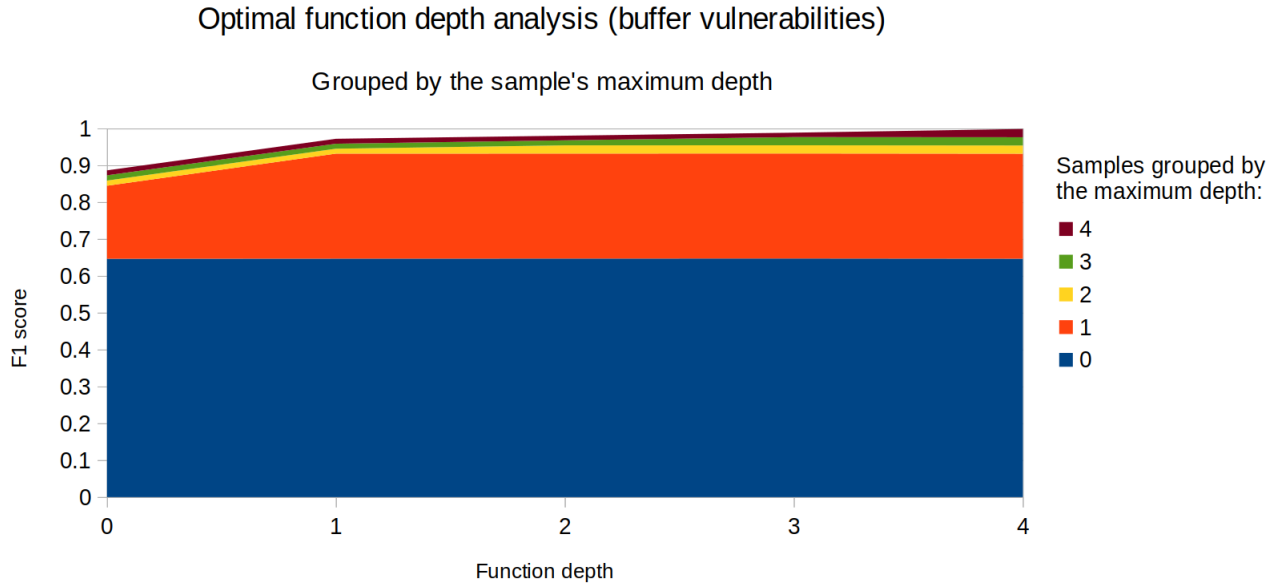

Figure 22: F1 score grouped by the sample's maximum depth.

	Depth: 0-1	Depth: 1-2	Depth: 2-3	Depth: 3-4
Sample's maximum depth: 0	0%	0%	0%	-0.1%
Sample's maximum depth: 1	**+8.7%**	0%	0%	0%
Sample's maximum depth: 2	-0.1%	**+0.9%**	0%	0%
Sample's maximum depth: 3	-0.1%	0%	**+0.9%**	0%
Sample's maximum depth: 4	0%	-0.1%	-0.1%	**+1.1%**

Table 8: Data set split by the maximum depth. The percentage indicates how the F1 score changes as the target depth increases.

5.3. COMPARING GLICE AGAINST FUNDED

This section addresses research question three: how does the GLICE model compare to the original FUNDED model in training duration and accuracy? To determine if GLICE (at function depth 4) outperforms the original FUNDED model we compare the training duration and accuracy. Although model training is a one-time cost, retraining may be necessary to adapt the model to changes in the security landscape. These changes include the emergence of new vulnerability types or the availability of new data samples. If model training takes longer, the associated costs are also higher. In case the model is trained in the cloud, the cloud provider often charges per hour and training on an on-premise machine requires a GPU which consumes electricity.

In addition to training time, accuracy is also an important metric. A common metric to measure accuracy is the F1 score. Ideally, we want high recall and high precision. The problem with a low recall is that the model does not find all the vulnerabilities in the target program. A model with a low precision introduces the opposite problem, not all samples classified as vulnerable are correct. A high recall and high precision ensure that most vulnerabilities will be discovered and less time is spend filtering out false positives.

First, we compare the training duration of our best-performing GLICE model against FUNDED. Training duration depends on multiple factors. These factors include the computing power (CPU, GPU, memory) available, the hyperparameters, the computational complexity of one epoch, and the total number of epochs until convergence.

Model training was performed on a single desktop PC. This desktop PC has a RTX 3060 GPU with 12GB of dedicated memory and 3584 CUDA cores. The CPU used is an AMD Ryzen 5 3600 and the system has 32GB of memory.

Due to constraints in hardware we were unable to optimize the hyperparameters of FUNDED or GLICE for our new data set. That is why we used the same hyperparameters as reported by the authors of FUNDED. The advantage of using the same hyperparameters for both FUNDED and GLICE is that the underlying neural networks have the same expressive power. This ensures that a difference in performance must be related to the program slicing algorithm or the new graph structure. The hyperparameters used are described in Appendix B.

To determine the computational complexity of one epoch, we calculated the number of samples our desktop PC can process per second during training. Dividing the total number of training samples by the samples processed per second, gives an indication how many seconds one epoch takes.

Figure 23 compares the models based on the number of samples processed per second during training. The more samples the model can process per second, the shorter an epoch lasts. First, we compare FUNDED against GLICE without program slicing (depth: 0). Both models analyze a single function to determine if the code is vulnerable. However, GLICE (depth: 0) is 59% faster than FUNDED. It can process 926 graphs per second instead of 582. This difference in performance can be explained by the fact that we removed the CDT graph from GLICE, reducing the number of nodes/edges. If we would also remove the CDT graph in the FUNDED model, the performance would be equal.

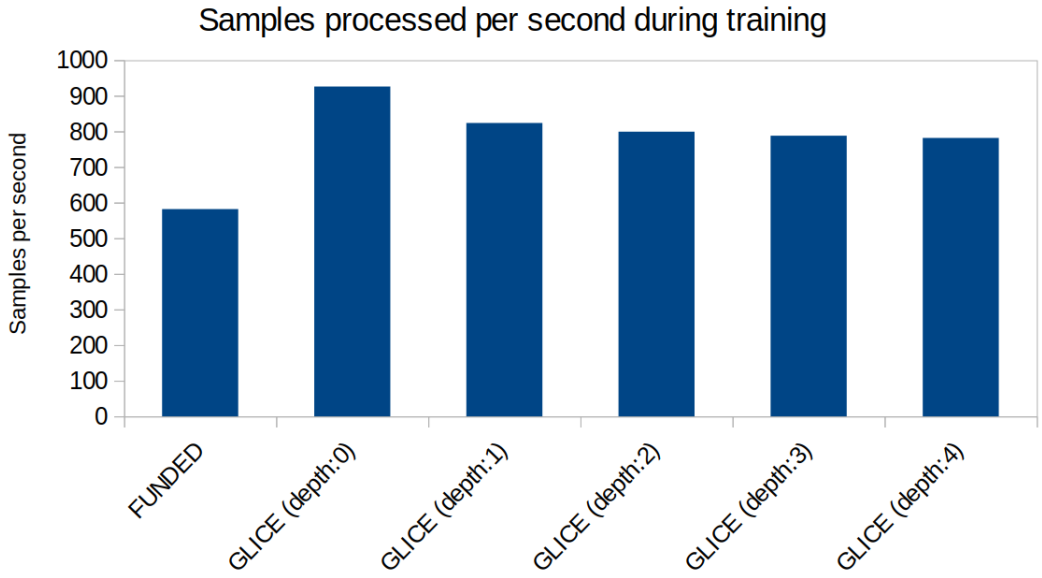

Figure 23: Comparison of the number of samples processed per second by FUNDED and GLICE (depth: 0-4).

If we compare GLICE at depth 0 against the best-performing depth of 4, we do notice a drop of 16% in speed. We conclude that increasing the size of the graph by including more functions has a negative impact on speed. However, our best-performing model (depth: 4) still outperforms FUNDED by 34%.

The samples processed per second by the model gives an indication of how long one epoch takes to compute. However, training the model for one epoch does not result in a highly accurate model. The neural network has to train multiple epochs to reach its full potential. This means that a model that processes more samples per second may run longer if the optimal number of epochs is higher. To compare FUNDED with GLICE, we calculated the average number of epochs it takes to reach 86% accuracy. This percentage was chosen because it is the minimum accuracy achieved by FUNDED.

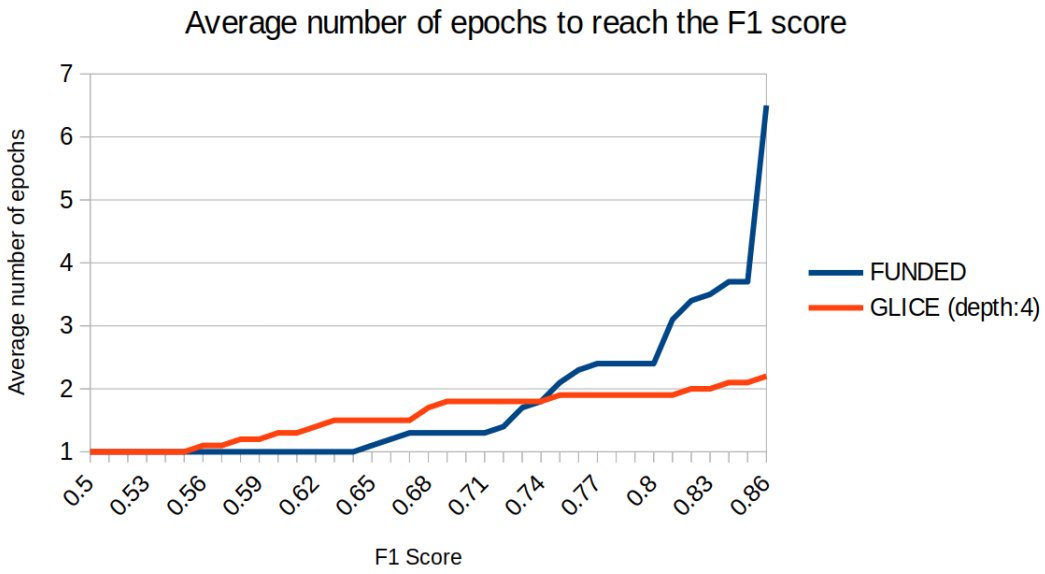

Figure 24: Number of epochs to reach a certain F1 score.

Figure 24 shows the average number of epochs it takes to reach a F1 score of 50% to 86%. First we compare the range 50% to 74%. We notice that FUNDED learns faster on average in this starting period. Next, we compare the range 74% to 86%. In this range GLICE is able to learn more quickly. Moreover, GLICE reaches the same accuracy as FUNDED (86%) in an almost 3x shorter time period.

Finally, we compare the accuracy of FUNDED versus GLICE (at depth 4). Figure 25 shows the difference in precision, recall, and the F1 score. The recall does not change significantly, but the precision increases by 21%. The improved precision results in a 0.12 increase in F1 score because this metric is the harmonic mean of the precision/recall. Furthermore, when we compare the F1 score of FUNDED against GLICE at depth 0 through 3 we notice that GLICE outperforms FUNDED in all four cases.

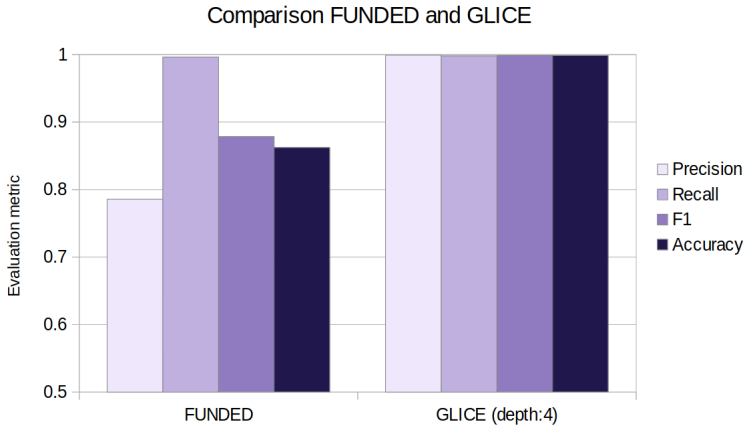

Figure 25: Comparison FUNDED vs GLICE (depth: 4).

6. RELATED WORK

Vulnerability detection using deep learning is an active research field. One of the first studies in this field was VulDeePecker [12]. They combined BLSTM (Bidirectional Long Short-Term Memory) neural networks and program slicing to find vulnerabilities. A follow up study called SySeVR extended their program slicing algorithm with control dependence information [13].

Our work is inspired by the program slicing algorithm described in VulDeePecker and SySeVR. Applying program slicing for vulnerability detection is therefore not a new idea. The distinguishing factor of our work is that we provide evidence of why this technique is so essential. Previous work gave no indication that program slicing would yield a more accurate model. In our work, we provided evidence that in certain scenarios program slicing can outperform function-level analysis in terms of accuracy.

In addition to providing this evidence, we also made two key contributions to the program slicing algorithm. The first contribution is that we included more language constructs in the algorithm. Prior work was limited to lines of code defined in a function/method. We extended the algorithm by also including global variables, function pointer calls, macros, and structures in the resulting slice. These improvements led to more security-relevant information being included in the slices.

Another key contribution was preserving the graph structure of the program slice. The network architecture used by VulDeePecker/SySeVR operated on a sequence of tokens. In our work, we used a graph neural network. That is why we not only needed to preserve the tokens (also known as nodes) but also the edges between these tokens. These edges included the parent-child (AST), control flow, data dependence, and control dependence relationships. The main benefit of including these relationships is that the neural network can analyze control/data flow to find better vulnerability patterns.

Our work also has a lot in common with Devign [29] and FUNDED [22]. The common factor is the use of graph neural networks to detect software vulnerabilities. Devign was the first to introduce this idea and FUNDED enhanced their work. There enhancements mainly included additional edge types, such as control dependence. However, both Devign and FUNDED were limited to detecting vulnerabilities at the function-level. We hypothesized that this is a major limitation because vulnerabilities can occur across function boundaries. To test this hypothesis, we extended FUNDED with the program slicing algorithm. Therefore, the key distinguishing factor of our work is the inclusion of this algorithm. In addition, we also made a smaller contribution in the form of an improved word embedding model for FUNDED.

7. DISCUSSION

The objective of our research was to improve an existing vulnerability detection model by introducing program slicing. We hypothesized that program slicing leads to better predictions than function-level analysis. The rationale for this is that vulnerabilities can occur across function boundaries and program slicing captures the interaction between functions.

The results indicate that program slicing indeed can outperform function-level analysis in certain scenarios. In our data set, more than one third of the vulnerabilities occur across function boundaries. As a result, the model's accuracy improves as we increase the function depth analyzed by the program slicing algorithm. This shows that not only the model's neural network architecture is relevant but also the input we feed it.

However, it is important to note that our results do not guarantee that program slicing will always be superior in terms of accuracy. The results apply only to buffer vulnerabilities primarily originating from SARD. Samples originating from SARD are known to be simpler to detect than real-world vulnerabilities. Another drawback is that these samples are generated using templates, causing samples to be similar.

In addition, the results indicate that our model using program slicing is less computationally expensive than the baseline model using function-level analysis. This is supported by the fact that our model is able to achieve the same accuracy as FUNDED in a shorter amount of time. However, this conclusion depends on the function depth used. In our data set, the samples had a maximum depth of four. That is why we did not analyze beyond four. However, if our data set contained more real-world samples the depth may need to be higher. Increasing the depth requires more computing power as shown in Figure 23. Therefore, in practice it will probably be a trade-off between accuracy and computational efficiency.

Finally, the results show that our program slicing implementation contains more security-relevant lines of code than VulDeePecker/SySeVR. We measured this by comparing the lines of code included in the slice against the meta information provided by SARD. This meta-information was available in the form of comments marking (non-)vulnerable lines. Our algorithm contains more security-relevant lines because it also analyzes function pointer calls and global variables.

However, there is a trade-off between completeness and soundness. In the context of slicing, a sound algorithm would return only lines that effect the criterion. On the other hand, a complete algorithm would return all lines that effect the criterion. The disadvantage of soundness is that it misses some lines and the disadvantage of completeness is that it may return too many lines. We believe our algorithm is more complete but this may be at the cost of soundness.

8. Conclusion & Future work

8.1. Conclusion

Our research efforts were aimed at measuring how program slicing can improve a graph neural network's ability to detect buffer vulnerabilities. To reach this goal, three research questions were formulated. The first research question addressed the design of the program slicing algorithm. The main requirement for this algorithm was that lines of code related to buffer vulnerabilities are preserved.

By performing a literature review, we discovered existing program slicing algorithms called VulDeePecker and SySeVR. We evaluated their algorithms by validating that all vulnerable lines of code were included in the resulting slices. We noticed that 6% of the vulnerable lines were missing. By including four missing language constructs in the algorithm, all vulnerable lines in our data set were included. The missing constructs were global variables, function pointer calls, macros, and structures.

The second question was concerned with the optimal target depth of our program slicing algorithm. This parameter determines the upper limit allowed for a function's depth in the call tree. We hypothesized that increasing the target depth will improve the overall accuracy of the model. However, we believed that increasing it beyond a certain point reduces the models ability to extract patterns because the graph becomes too large.

We confirmed the first hypothesis that increasing the function depth improves the accuracy. As a result, the highest function depth resulted in the best performing model with a F1 score of 99.9%. No evidence was found for our earlier statement that the model cannot extract patterns from larger graphs. It is important to note that these results are limited to our buffer vulnerability data set mainly originating from SARD.

By comparing this best performing model against the original function-level approach we noticed an increase in F1 score by 12%. This confirms our believe that program slicing can detect vulnerabilities across function boundaries that a function-level approach cannot. Finally, we compared training time. Our GLICE model was able to complete one epoch 34% faster and needed less epochs to reach a similar level of performance as the FUNDED model.

8.2. Future work

In this section, we describe the next steps that should be taken to advance the field of deep learning and vulnerability detection.

One of the next steps is extending our data set. Currently, the data set contains only buffer vulnerabilities. This can be expanded to other types of vulnerabilities to validate if these types also benefit from program slicing. Also the percentage of real-world samples from the NVD should be increased to prove its practical applicability. The SARD samples may be too predictable because they are based on templates. Another issue with the SARD data set is that it has not been updated since Oct. 2017. This means that the data set does not contain language constructs from newer ANSI standards.

An alternative research direction is understanding why our model makes certain predictions. By using explanation methods designed for graph neural networks, we can determine if the discovered patterns are actually useful. The work of Ying et al. [27] called GNN Explainer can be used to indicate the relevance of edges in a prediction. Using these relevant edges, a subgraph can be constructed that contains the most relevant features.

Another research area that can be considered in future work is adversarial attacks. In an adversarial attacks a deep-learning model is fooled to give an incorrect prediction. This is done by modifying the input data in such a way that the model misclassifies it. In the context of vulnerability detection, an attacker could modify a vulnerable code snippet so that it is marked as non-vulnerable. This enables an attacker to introduce malicious code in a software project. The work by Dai et al. [5] could be used to generate modified graph structures that fool the graph neural network. Defensive techniques should be implemented to prevent adversarial attacks.

Finally, automatically patching vulnerabilities can be considered. Deep-learning models are much faster at finding vulnerabilities than human experts. As deep-learning models become more accurate, the number of vulnerabilities discovered in real-world software will rise. There may not be enough human experts to fix these issues in a reasonable time. To combat this issue, neural networks should be able to both detect and patch vulnerabilities.

LITERATURE

[1] F. E. Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.

[2] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, 1976.

[3] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation*, pages 290–301, 1994.

[4] B. Chess and J. West. *Secure programming with static analysis*. Pearson Education, 2007.

[5] H. Dai, H. Li, T. Tian, X. Huang, L. Wang, J. Zhu, and L. Song. Adversarial attack on graph structured data. In *International conference on machine learning*, pages 1115–1124. PMLR, 2018.

[6] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose. Automatic feature learning for vulnerability prediction. *arXiv preprint arXiv:1708.02368*, 2017.

[7] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 69–80, 2003.

[8] A. M. Fout. *Protein interface prediction using graph convolutional networks*. PhD thesis, Colorado State University, 2017.

[9] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[10] A. K. Jain, J. Mao, and K. M. Mohiuddin. Artificial neural networks: A tutorial. *Computer*, 29(3):31–44, 1996.

[11] M. Kuhn, K. Johnson, et al. *Applied predictive modeling*, volume 26. Springer, 2013.

[12] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[13] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 2021.

[14] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[15] G. McGraw. Software security. *IEEE Security & Privacy*, 2(2):80–83, 2004.

[16] V. H. Nguyen and L. M. S. Tran. Predicting vulnerable software components with dependency graphs. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, pages 1–8, 2010.

[17] R. E. Noonan. An algorithm for generating abstract syntax trees. *Computer Languages*, 10(3-4):225–236, 1985.

[18] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. *ACM Sigplan Notices*, 19(5):177–184, 1984.

[19] R. T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 133–138, 1959.

[20] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pages 757–762. IEEE, 2018.

[21] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[22] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security*, 2020.

[23] M. Weiser. Program slicing. *IEEE Transactions on software engineering*, (4):352–357, 1984.

[24] Y. Wu, D. Lian, Y. Xu, L. Wu, and E. Chen. Graph convolutional networks with markov random field reasoning for social spammer detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1054–1061, 2020.

[25] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604. IEEE, 2014.

[26] G. Ye, Z. Tang, H. Wang, D. Fang, J. Fang, S. Huang, and Z. Wang. Deep program structure modeling through multi-relational graph-based learning. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 111–123, 2020.

[27] R. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec. Gnn explainer: A tool for post-hoc explanation of graph neural networks. *arXiv preprint arXiv:1903.03894*, 2019.

[28] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[29] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *arXiv preprint arXiv:1909.03496*, 2019.

WEB LINKS

[30] M. Brockschmidt. Tensorflow 2 library implementing graph neural networks. URL https://github.com/microsoft/tf2-gnn.

[31] M. Fey. Graph neural networks. URL https://www.youtube.com/watch?v=bgKScUgAyvM.

[32] I. I. O. for Standardization. C17. URL https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2310.pdf.

[33] P. S. Foundation. Python. URL https://www.python.org/.

[34] Google. Tensorflow. URL https://www.tensorflow.org/.

[35] S. Mishra. Biden administration declares emergency over fuel pipeline hack and warns more attacks will come, 2021. URL https://www.independent.co.uk/news/world/americas/us-politics/cyber-attack-us-pipeline-biden-b1844775.html.

[36] MITRE. Cwe - common weakness enumeration. URL https://cwe.mitre.org/.

[37] MITRE. Cwe-121: Stack-based buffer overflow, 2006. URL https://cwe.mitre.org/data/definitions/121.html.

[38] MITRE. Cwe-122: Heap-based buffer overflow, 2006. URL https://cwe.mitre.org/data/definitions/122.html.

[39] MITRE. Cwe-124: Buffer underwrite ('buffer underflow'), 2006. URL https://cwe.mitre.org/data/definitions/124.html.

[40] MITRE. Cwe-125: Out-of-bounds read, 2006. URL https://cwe.mitre.org/data/definitions/125.html.

[41] MITRE. Cwe-126: Buffer over-read, 2006. URL https://cwe.mitre.org/data/definitions/126.html.

[42] MITRE. Cwe-127: Buffer under-read, 2006. URL https://cwe.mitre.org/data/definitions/127.html.

[43] MITRE. Cwe-787: Out-of-bounds write, 2009. URL https://cwe.mitre.org/data/definitions/787.html.

[44] MITRE. 2021 cwe top 25 most dangerous software weaknesses, 2021. URL https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html.

[45] N. N. I. of Standards and Technology). Nvd (national vulnerability database), . URL https://nvd.nist.gov/.

[46] N. N. I. of Standards and Technology). Sard (software assurance reference dataset), . URL https://samate.nist.gov/SARD/.

[47] U. G. P. Office. America is under cyber attack. URL https://www.govinfo.gov/content/pkg/CHRG-112hhrg77380/html/CHRG-112hhrg77380.htm.

[48] W. Turton and K. Mehrotra. Hackers breached colonial pipeline using compromised password, 2021. URL https://www.bloomberg.com/news/articles/2021-06-04/hackers-breached-colonial-pipeline-using-compromised-password.

[49] H. Wang. Funded. URL https://github.com/HuantWang/FUNDED_NISL.

A. EXPERIMENTAL RESULTS

Fold	Precision	Recall	F1	Accuracy
Reproduced model (fold 0)	0.8583	0.9776	0.914	0.9081
Reproduced model (fold 1)	0.861	1	0.9253	0.9193
Reproduced model (fold 2)	0.8548	0.9507	0.9002	0.8946
Reproduced model (fold 3)	0.861	1	0.9253	0.9193
Reproduced model (fold 4)	0.861	1	0.9253	0.9193
Reproduced model (fold 5)	0.861	1	0.9253	0.9193
Reproduced model (fold 6)	0.8627	0.9865	0.9205	0.9148
Reproduced model (fold 7)	0.8594	0.9596	0.9068	0.9013
Reproduced model (fold 8)	0.8599	0.991	0.9208	0.9148
Reproduced model (fold 9)	0.8577	0.9462	0.8998	0.8946
Reproduced model (average of 10-fold)	0.86	0.98	0.92	0.91
Results original paper (average of 10-fold)	0.83	1.00	0.92	0.90

Table 9: Comparison reproduced model against results reported in the original paper (CWE-78: OS Command Injection)

Fold	FUNDED (Joern + CDT)	FUNDED (Joern)	GLICE (no slicing)	GLICE (depth:0)	GLICE (depth:1)	GLICE (depth:2)	GLICE (depth:3)	GLICE (depth:4)
Fold 0	0.8821	0.8839	0.8982	0.8945	0.9675	0.98	0.9868	0.9995
Fold 1	0.8782	0.8878	0.9021	0.9042	0.9735	0.9825	0.9896	0.998
Fold 2	0.8725	0.8736	0.8938	0.889	0.97	0.9735	0.9868	0.9985
Fold 3	0.8791	0.8849	0.896	0.8942	0.9657	0.9791	0.9873	0.9995
Fold 4	0.878	0.8749	0.8917	0.8923	0.9726	0.9815	0.9882	0.9985
Fold 5	0.8844	0.8821	0.8912	0.8974	0.9707	0.9824	0.9876	0.999
Fold 6	0.8753	0.8758	0.8885	0.8911	0.9735	0.9816	0.9921	0.9985
Fold 7	0.8849	0.8839	0.8938	0.8946	0.9725	0.9801	0.9862	0.999
Fold 8	0.8701	0.8714	0.8891	0.8886	0.966	0.9758	0.9853	0.996
Fold 9	0.8798	0.881	0.8968	0.8982	0.9773	0.9784	0.9888	0.999
Average 10-fold	0.8784	0.8799	0.8941	0.8944	0.9709	0.9795	0.9879	0.9986

Table 10: F1 score of deep-learning models (buffer vulnerabilities)

Fold	FUNDED (Joern + CDT)	FUNDED (CDT)	GLICE (no slicing)	GLICE (depth:0)	GLICE (depth:1)	GLICE (depth:2)	GLICE (depth:3)	GLICE (depth:4)
Fold 0	0.7934	0.792	0.8172	0.8171	0.9505	0.9673	0.9805	1
Fold 1	0.7896	0.7983	0.8277	0.8272	0.9519	0.9684	0.9892	0.998
Fold 2	0.7756	0.7773	0.8153	0.8106	0.9499	0.9687	0.9796	1
Fold 3	0.7904	0.7948	0.8122	0.8133	0.9729	0.9646	0.9768	1
Fold 4	0.7831	0.7783	0.8097	0.8088	0.9475	0.9656	0.9805	0.999
Fold 5	0.7953	0.7891	0.8122	0.8152	0.9432	0.9738	0.9891	0.998
Fold 6	0.7789	0.7791	0.8098	0.8074	0.9485	0.9647	0.9854	0.999
Fold 7	0.7936	0.7932	0.8093	0.8106	0.95	0.9628	0.9824	0.999
Fold 8	0.77	0.7728	0.8016	0.8014	0.9369	0.9564	0.9795	0.999
Fold 9	0.7866	0.788	0.8183	0.8165	0.9565	0.9717	0.9787	0.999
Average 10-fold	0.7857	0.7863	0.8133	0.8128	0.9508	0.9664	0.9822	0.9991

Table 11: Precision score of deep-learning models (buffer vulnerabilities)

Fold	FUNDED (Joern + CDT)	FUNDED (Joern)	GLICE (no slicing)	GLICE (depth:0)	GLICE (depth:1)	GLICE (depth:2)	GLICE (depth:3)	GLICE (depth:4)
Fold 0	0.9931	1	0.997	0.9882	0.9852	0.9931	0.9931	0.999
Fold 1	0.9891	1	0.9911	0.997	0.9961	0.997	0.9901	0.998
Fold 2	0.997	0.997	0.9891	0.9842	0.9911	0.9783	0.9941	0.997
Fold 3	0.9901	0.998	0.999	0.9931	0.9585	0.9941	0.998	0.999
Fold 4	0.999	0.999	0.9921	0.9951	0.999	0.998	0.996	0.998
Fold 5	0.996	1	0.9872	0.998	1	0.9911	0.9862	1
Fold 6	0.999	1	0.9842	0.9941	1	0.999	0.999	0.998
Fold 7	1	0.998	0.998	0.998	0.996	0.998	0.9901	0.999
Fold 8	1	0.999	0.998	0.997	0.997	0.996	0.9911	0.9931
Fold 9	0.998	0.999	0.9921	0.998	0.999	0.9852	0.999	0.999
Average 10-fold	0.9961	0.999	0.9928	0.9943	0.9922	0.9930	0.9937	0.9980

Table 12: Recall score of deep-learning models (buffer vulnerabilities)

Fold	FUNDED (Joern + CDT)	FUNDED (Joern)	GLICE (no slicing)	GLICE (depth:0)	GLICE (depth:1)	GLICE (depth:2)	GLICE (depth:3)	GLICE (depth:4)
Fold 0	0.8672	0.8687	0.887	0.8835	0.9669	0.9798	0.9867	0.9995
Fold 1	0.8628	0.8736	0.8924	0.8944	0.9729	0.9822	0.9896	0.998
Fold 2	0.8543	0.8558	0.8825	0.8771	0.9694	0.9733	0.9867	0.9985
Fold 3	0.8638	0.8702	0.884	0.8825	0.9659	0.9788	0.9872	0.9995
Fold 4	0.8612	0.8572	0.8794	0.8799	0.9718	0.9812	0.9881	0.9985
Fold 5	0.8699	0.8665	0.8794	0.8859	0.9699	0.9822	0.9876	0.999
Fold 6	0.8577	0.8582	0.8765	0.8785	0.9728	0.9812	0.9921	0.9985
Fold 7	0.87	0.869	0.8814	0.8824	0.9718	0.9797	0.9862	0.999
Fold 8	0.8507	0.8527	0.8755	0.875	0.9649	0.9753	0.9852	0.996
Fold 9	0.8636	0.8651	0.8859	0.8869	0.9768	0.9783	0.9886	0.999
Average 10-fold	0.8621	0.8637	0.8824	0.8826	0.9703	0.9792	0.9878	0.9986

Table 13: Accuracy score of deep-learning models (buffer vulnerabilities)

Sample's maximum depth	GLICE (depth:0)	GLICE (depth:1)	GLICE (depth:2)	GLICE (depth:3)	GLICE (depth:4)
0	0.6464	0.6468	0.647	0.6471	0.6466
1	0.1981	0.2849	0.2849	0.2849	0.2849
2	0.0141	0.0133	0.0223	0.0223	0.0223
3	0.0142	0.0136	0.0137	0.0223	0.0223
4	0.0134	0.0134	0.0124	0.0118	0.0223

Table 14: Accuracy split by depth

FUNDED (Joern + CDT)	GLICE (depth:0)	GLICE (depth:1)	GLICE (depth:2)	GLICE (depth:3)	GLICE (depth:4)
581.79	925.87	823.68	799.26	787.89	781.57

Table 15: Number of graphs processed per second during training.

F1	FUNDED	GLICE (depth:4)
0.5	1	1
0.51	1	1
0.52	1	1
0.53	1	1
0.54	1	1
0.55	1	1
0.56	1	1.1
0.57	1	1.1
0.58	1	1.2
0.59	1	1.2
0.6	1	1.3
0.61	1	1.3
0.62	1	1.4
0.63	1	1.5
0.64	1	1.5
0.65	1.1	1.5
0.66	1.2	1.5
0.67	1.3	1.5
0.68	1.3	1.7
0.69	1.3	1.8
0.7	1.3	1.8
0.71	1.3	1.8
0.72	1.4	1.8
0.73	1.7	1.8
0.74	1.8	1.8
0.75	2.1	1.9
0.76	2.3	1.9
0.77	2.4	1.9
0.78	2.4	1.9
0.79	2.4	1.9
0.8	2.4	1.9
0.81	3.1	1.9
0.82	3.4	2
0.83	3.5	2
0.84	3.7	2.1
0.85	3.7	2.1
0.86	6.5	2.2

Table 16: Number of epochs to reach the F1 score.

B. HYPERPARAMETERS

Hyperparameter	Value
Max graphs per batch	128
Add self loop edges	True
Tie forward/backward edges	True
GNN aggregation function	sum
GNN message activation function	ReLU
GNN hidden dim	256
GNN number of edge MLP hidden layers	1
GNN initial node representation activation	tanh
GNN dense intermediate layer activation	tanh
GNN number of layers	5
GNN dense every num layers MLP hidden layers	10000
GNN residual every number of layers	2
GNN layer input dropout rate	0.2
GNN global exchange mode	gru
GNN global exchange every num layers	10000
GNN global exchange number of heads	4
GNN global exchange dropout rate	0.2
Optimizer	Adam
Learning rate	0.001
Graph aggregation number of heads	16
Graph aggregation hidden layers	[128]
Graph aggregation dropout rate	0.2