Neural Computing and Applications (2024) 36:15869–15891
https://doi.org/10.1007/s00521-024-09819-3

ORIGINAL ARTICLE

Multi-class vulnerability prediction using value flow and graph neural networks

Connor McLaughlin[1,2] · Yi Lu[1,2]

Received: 21 March 2023 / Accepted: 15 April 2024 / Published online: 20 May 2024

Abstract
In recent years, machine learning models have been increasingly used to detect security vulnerabilities in software, due to their ability to achieve high performance and lower false positive rates compared to traditional program analysis tools. However, these models often lack the capability to provide a clear explanation for why a program has been flagged as vulnerable, leaving developers with little reasoning to work with. We present a new method which not only identifies the presence of vulnerabilities in a program, but also the specific type of error, considering the whole program rather than just individual functions. Our approach utilizes graph neural networks that employ inter-procedural value flow graphs, and instruction embedding from the LLVM Intermediate Representation, to predict a class. By mapping these classes to the Common Weakness Enumeration list, we provide a clear indication of the security issue found, saving developers valuable time which would otherwise be spent analyzing a binary vulnerable/non-vulnerable label. To evaluate our method's effectiveness, we used two datasets: one containing memory-related errors (out of bound array accesses), and the other a range of vulnerabilities from the Juliet Test Suite, including buffer and integer overflows, format strings, and invalid frees. Our model, implemented using PyTorch and the Gated Graph Sequence Neural Network from Torch-Geometric, achieved a precision of 96.35 and 91.59% on the two datasets, respectively. Compared to common static analysis tools, our method produced roughly half the number of false positives, while identifying approximately three times the number of vulnerable samples. Compared to recent machine learning systems, we achieve similar performance while offering the added benefit of differentiating between classes. Overall, our approach represents a meaningful improvement in software vulnerability detection, providing developers with valuable insights to better secure their code.

Keywords Graph neural networks · Knowledge representation · Software security · Program analysis

1 Introduction

One of the most common causes of security vulnerabilities found in computer systems is software bugs, or errors. These errors cause programs to behave incorrectly, indirectly leading to data loss, system compromise and damage. These bugs are usually disclosed as security advisories, and published in databases such as the MITRE Common Vulnerabilities and Exposures (CVE) List [1], or the National Vulnerability Database (NVD) [2]. Analysis of yearly CVE disclosures has shown significant growth in the number of advisories published in the last 5 years, and no signs of reduction [3], with thousands of new vulnerabilities being discovered every month. Of greater concern is the number of zero-day attacks in the wild, with Rapid7 finding there has been a significant increase from the years 2020 to 2021 alone [4]. A zero-day attack is where an unpatched vulnerability in software is being actively exploited by malicious actors, leading to a race between both the software developers and IT professionals to patch/protect systems before significant damage can be caused.

Early detection of vulnerabilities is critical for reducing the frequency of these zero-day attacks, with bugs ideally

✉ Connor McLaughlin
connor.mclaughlin@hdr.qut.edu.au

Yi Lu
yt.lu@qut.edu.au

[1] Queensland University of Technology, Brisbane, QLD, Australia

[2] Cyber Security Cooperative Research Centre, Joondalup, Australia

being identified prior to software shipping. Traditionally, this has been done through a combination of static program analysis, manual auditing performed by humans, and fuzzing/genetic-based systems. Each of these approaches has both upsides and downsides, for example, static analysis usually results in high false positive rates and sub-optimal detection rates [5], and fuzzing can have extremely long time intervals between error discovery [6]. In the literature, machine learning has overtaken traditional software analysis methods for vulnerability detection, with Hanif et al. finding that 74% of publications were using some form of machine learning [7], in particular, deep learning, being the most popular.

Earlier work using machine learning for bug detection dates back to 2012, where Hovsepyan et al. utilized text analysis techniques, combined with a support vector machine for detecting errors [8]. Other approaches include Long Short-Term Memory-based gadget classification of intra-procedural slices [9], as well as convolutional neural networks based on natural language processing (NLP) techniques, both at the source code level [10, 11], and assembly/instruction level [12]. However, we believe that using NLP-based methods significantly increases the difficulty of the problem. Not only does the classifier have to learn the characteristics of the errors it is trying to detect, but it also has to learn the semantics of the language it is analyzing. Furthermore, the detection system requires a large enough training set, particular to the language being analyzed, and a model trained on one language is not directly transferable to other languages.

Instead of detecting vulnerabilities in the program's code, whether it be source or binaries, we considered an alternative approach: Can we train a machine learning model to detect vulnerabilities in a value flow graph computed from the program? Using such a graph representation of the program would make the detection model less dependent on the original language that the program is written in, and its semantics, as instead we are examining where and how data are used in computation. In addition, we can detect vulnerabilities across function and file boundaries, as value flows are computed for the entire program, compared to analyzing individual functions in isolation. Having this greater amount of information can lead to higher precision for detection; however, it is a trade-off, as such performance is dependent on the model's ability to filter out flows irrelevant to the learning task.

Value flow graphs are a state-of-the-art representation of programs, incorporating both control and data flow, and alias-aware memory analysis. This makes them an ideal candidate for vulnerability detection applications, as combined with the aforementioned fact that value flows are computed interprocedurally, we can reason about the security of the whole program. The value flow graphs contain complex relationships, describing how data traverses the program, with larger programs containing tens of thousands of nodes and edges. These types of graphs are usually used for optimization and program analysis tasks, such as [13], and not machine learning applications. Graph neural networks are known for their ability to transform and capture attributes of both the data and relations contained within graphs [14], and paired with the value flow graphs, represent a natural fit for our task of software vulnerability detection.

Using neural networks for vulnerability detection provides many benefits, including the ability to learn new types of vulnerabilities through data, rather than having to manually identify the characteristics of the vulnerability, and write rules which systems can then apply. Compared to runtime/coverage-based analysis, such as fuzzing, where non-trivial configuration time is required, machine learning systems can analyze programs with minimal effort from the user. However, as a result, some flexibility is lost, as many of these machine learning systems are trained to simply identify the presence of vulnerabilities, rather than reporting the specific type of error. While identification, or binary classification is still valuable, it still requires an expert to identify the exact type of error and propose a solution. For this reason, we chose to build a multi-class classification system, which not only identifies vulnerabilities but reports the types of errors as well, saving human time by reducing the range of errors that must be considered. However, it is a more difficult classification problem, as the model must now identify which type of weaknesses the features of the error most closely align with. Despite the challenges, our research suggests that multi-class machine learning systems can achieve high performance, exceeding that of rule-based systems, which are also capable of identifying multiple classes/vulnerability types.

2 Our contributions

In this article, we present a new method for multi-class vulnerability detection in programs, utilizing static, inter-procedural value flow graphs and rich instruction embedding as a program representation, and a graph neural network for classification. Figure 1 shows a high-level overview of the process for determining which vulnerability a sample contains. Our method begins by pre-processing the input program (Sect. 5, which includes computing the value flow graph and transforming said graph into a form suitable for input into a neural network. In addition to the graph itself, we also compute an embedding from both the graph node attributes, as well as the original instruction which the graph nodes were derived from (Sect. 5.3). Our machine learning model predicts a

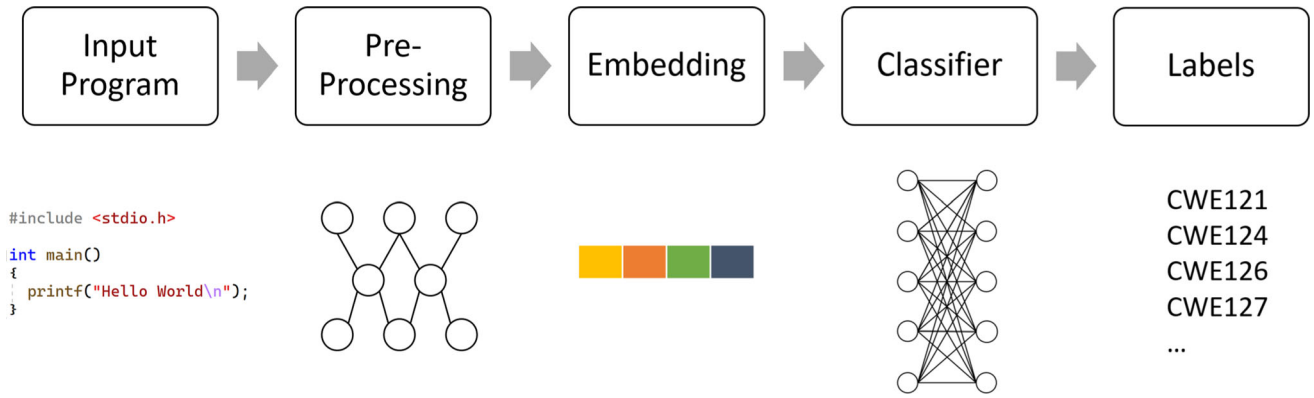


Fig. 1 High-level overview of method

multi-label category from the sample's graph and embeddings, suggesting which vulnerabilities the sample may contain, based on a probability for each class. We evaluated the model's performance by:

- Generating two datasets for evaluating our model: a synthetic dataset with simple memory-related errors, using the Csmith program generation tool [15], as well as a subset of the Juliet Test suite [16]. We chose to focus on memory errors due to the high percentage of security vulnerabilities being related to some kind of memory safety issues [17, 18], and value flow graphs providing a means of capturing the patterns that lead to these types of errors [19].
- Benchmarking our model in multi-class vulnerability detection, identifying and reasoning about the areas of weakness.
- Benchmarking our model against alternative machine learning systems and static analysis tools. As these systems only produce binary classifications, we had to also benchmark our model in binary mode to enable comparison.
- Examining the impact of the sub-features in the node embedding, determining which components are most critical for the downstream learning task, and how much value is added by providing the model with more concise detail in the input, rather than relying on learning alone.

3 Background

3.1 Inter-procedural security vulnerabilities

As mentioned in Sect. 1, errors in program code, or bugs, are one of the most common causes of security vulnerabilities in computer software. The CWE list is a "list of software and hardware weakness types" [1] which groups these security vulnerabilities into categories, based on the characteristics of the bug itself. These categories include memory errors, such as out-of-bounds reads/writes, integer overflow/underflow, uncontrolled format string, and hundreds of others [1].

Figure 2 shows an example of an inter-procedural buffer overflow error. We allocate storage for an array of 10 integers, using the `malloc` function provided by the C library, call `func_b`, which stores 10 integers to this array, and then print out the first element in the array. Since the size parameter in `malloc` refers to the number of bytes of storage to allocate, not the number of elements to allocate, instead of allocating 40 bytes of storage, we are only allocating 10 bytes. In this case, the programmer should multiply the number of elements by the size of each element (`malloc(10 * sizeof(int))`, or use an alternative routine such as `calloc`, which also handles the case where the multiply results in an integer overflow. Such errors are very common vulnerabilities, with out-of-bounds writes being in the top 25 most dangerous software weaknesses [1].

```
void func_b();
int* func_a() {
    int* data = malloc(10);
    func_b(data);
    printf("%d\n", data[0]);
    return data;
}

void func_b(int* data) {
    int source[10] = {0};
    memcpy(data, source, 10 * sizeof
        (int));
}
```

Fig. 2 Example of inter-procedural buffer overflow error

With the above example, neither function on its own contains a buffer overflow error. While `func_a` allocates an incorrect number of bytes, no writes occur within the function itself. The overflow occurs in `func_b`, which assumes that the array parameter has at least 40 bytes of storage allocated. Therefore, the vulnerability exists in `func_b`, but only in the context of `func_a` calling it, making it an inter-procedural error. This makes considering each function in isolation insufficient for detecting these types of errors, as the label is dependent on the context. Following the example further, these two functions may not even be defined in the same file, or be contained within a different library which is resolved later at link time. With the large range of possibilities, we can argue that security is a property of the whole program, rather than something which we can reason about for individual functions, or even source files, highlighting the importance of using inter-procedural techniques for error detection.

3.2 Inter-procedural value flow graphs

Value flow analysis encompasses both the control and data dependencies of a program, in contrast to control or data flow graphs, which only consider one of these in isolation. We utilize SVF (Static Value-Flow Analysis Framework), a framework for scalable and precise inter-procedural static value flow analysis [19]. SVF is a state-of-the-art tool for analyzing C-like programs and computing value flows, by "leveraging recent advances in sparse analysis" [19]. The view of all value flows across a program forms the inter-procedural value flow graph, a representation of program dependencies produced by SVF analysis, and have previously been used for detecting memory leaks, uninitialized variables, security vulnerabilities, and tainted information flow [20–22] The inter-procedural nature of these graphs provides further benefit in a bug detection context, as security is a property of the entire program, not just one function, considering functions in isolation is insufficient for identifying some vulnerabilities.

Figure 3 shows a portion of the value flow graph generated for three functions in a simple C program, which stores the constant integer *42* to a pointer provided as a formal parameter, and copies it to another stack variable in a second function call. This example demonstrates SVF's ability to track program dependencies and data flow across procedures, in addition to the flow across memory, as the value is value being stored to stack and reloaded.

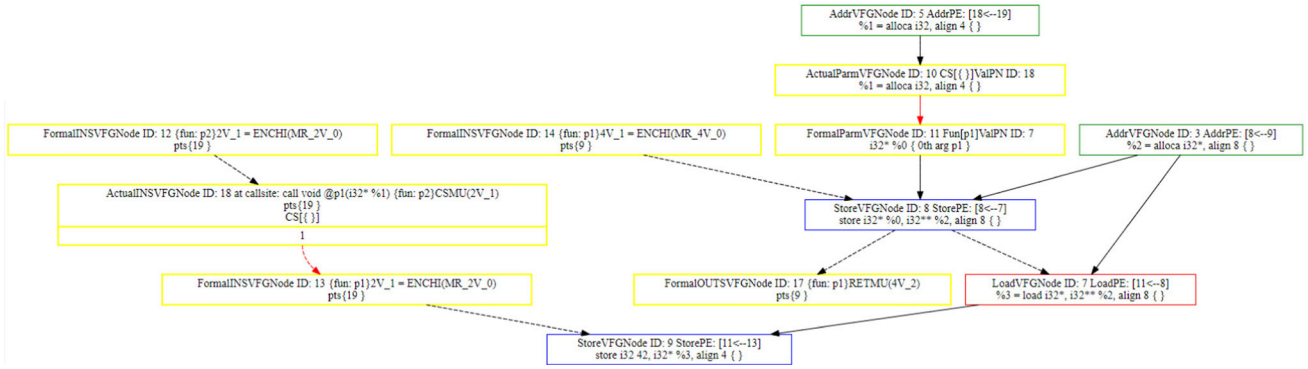
SVF uses Andersen's pointer analysis [23] to build a points-to set, which is used to derive define-use chains for variables where the address is taken, as well as top-level definitions, capturing alias-aware value flows. This method is key to SVF's ability to precisely analyze large programs in reasonable amounts of time, compared to more traditional iterative approaches [19]. The output of the analysis steps is combined with the program's IR-level instructions to produce a value flow graph, which can be utilized for the aforementioned applications. For example, points-to analysis can be also utilized to statically identify the memory buffers that each pointer references, and with the metadata from the buffer, identify whether a given access is safe. This also extends to other types of errors, such as numerical overflow, as the usages of all variables in the program can be traced through the graph, we can reason more accurately about the domain of operations, such as additions, and ignore locations when the operation is conclusively safe.

More recently, SVF has been utilized for semantic labeling and code captioning tasks (Flow2Vec, [24]), outperforming previous state-of-the-art approaches such as Code2Vec [25]. While Flow2Vec utilized higher-order proximity embedding to transform the value flow graph to a vector representation, in contrast to our proposed method of using Graph Neural Networks, this work highlights the value in using alias-aware program analysis for these kinds of tasks when compared to simpler methods such as AST paths, utilized by Code2Vec. SVF has also been used for vulnerability detection as recent as 2022, however, with a focus on binary classification, instead using a separate model trained for each type of weakness [21, 22].

SVF is built on the LLVM compiler framework [13], with the intermediate representation being used as the source for which the inter-procedural static value flow graph is computed. LLVM is a "a collection of modular and reusable compiler and toolchain technologies" [26]. Programs are compiled to the LLVM intermediate representation (IR), a strongly typed, single static assignment (SSA) bitcode format. LLVM provides a range of tools which operate on the intermediate representation, including analyzers and optimizers [27]. Code generation backends are available for many popular architectures, including x86, ARM, MIPS, RISC-V, and others. The LLVM framework, including the intermediate representation and backends are used for a range of languages, including C/C++, Rust, Swift, Zig, and others [26].

3.3 Graph neural networks

Graph neural networks have seen an increase in use over the last decade, as a method for performing learning and prediction type tasks on graph structured data, with a jump in popularity with the publication of *The Graph Neural Network Model* [28]. Types of data often represented in graph form include social media connections, citation networks, molecular sets, and encyclopedias [14]. Two of the common operations performed by these networks are node-based classification, where a label or class is




```
void p1(int* p) {
    *p = 42;
}
void p2(int* x, int* y) {
    *y = *x;
}
void p2() {
    int v1, v2;
    p1(&v1);
    p2(&v1, &v2);
}
```

Fig. 3 Subsection of the value flow graph produced for a simple program

predicted for every node in the graph based on its attributes and connections, and graph classification, where a label or class is predicted for the entire graph (considering all nodes and edges) [29].

The most relevant graph network model to our work is the Gated Graph Sequence Neural Network (GGSNN) [30]. The GGSNN produces a feature vector of user-specified length for each node in the graph, based on the attributes of the graph nodes, as well as the edges between them. The GGSNN has seen use for text-based reasoning applications, as well as program verification [30]. More recently, and most relevant, was its usage in the Devign vulnerability detection model [31]. Devign used the GGSNN with graphs representing programs (Abstract Syntax Trees/Control Flow Graphs/Data Flow Graphs) to learn to detect intra-procedural errors, and its success in this application was the main influence for our choice of this network design for our detection model.

4 Datasets

In this section, we describe how we generated and pre-processed the dataset, used to train and evaluate the vulnerability detection model. Two datasets were generated for our experiments; one fully synthetic focusing on out-of-bounds memory errors among complex control flow, and the second based on the well-studied Juliet Test Suite, covering a wider range of vulnerabilities.

4.1 Dataset A: fully synthetic/out-of-bounds errors

As the number of memory safety errors in security vulnerabilities is proportionately high [17, 18], our initial evaluation was performed on a synthetically generated dataset of memory errors. This allowed us to efficiently generate the large amounts of data required for training. We considered using data augmentation, but naïve approaches of mutating the input source code would not be usable in our application, as to compute the value flow graph, the input code must compile successfully.

We utilized the Csmith [15] test generation tool to generate these synthetic programs. Csmith "generates random C programs that statically and dynamically confirm to the C99 standard" [32]. It is useful for "stress testing compilers, static analyzers, and other tools that process C code" [32]. A range of options are supported, including injecting branches, loops, arrays, structs, and unions into the generated code. For this dataset, we decided to begin with simple, intra-procedural memory-based errors, and CSmith's option to inject out-of-bounds array accesses satisfied this criteria. When generating these programs, Csmith reports the number of out-of-bounds accesses

injected into the program, which we used to label the samples.

50,000 vulnerable and 50,000 non-vulnerable samples were generated for this dataset. Figure 4 shows a snippet of one of these samples, with the out-of-bounds access occurring for the first 1000 iterations of the loop (`l_4`).

4.2 Dataset B: Juliet test suite

To evaluate the model's performance on a wider range of bug types, we utilized the Juliet Test Suite. The Juliet Test Suite is packaged as a set of test cases, grouped by the CWE type, or vulnerability. In some cases, the example is limited only to a single source file, but in others, the example spans multiple source files/translation units. Prior experimentation has shown that considering these source files in isolation is insufficient for bug detection, and that analysis tools need to perform whole program analysis to accurately detect errors, which is a trait shared with real-world applications. Figure 5 shows an example of one of the vulnerable programs in the Juliet Test Suite, with comments removed and identifiers shortened for improving readability.

The program shown in Fig. 5 is an example of an inter-procedural, as well as cross-compilation-unit vulnerability. While the stack-based buffer overflow occurs in the second file/function (`badSink()`), whether any overflow occurs is context dependent, similar to the example shown in Sect. 3.1. The caller must ensure that the buffer provided is sufficient to store the `SRC_STRING` string, which is 11 characters in total, including the null terminator. This

```
static int32_t  func_1(void) {
    int32_t l_2[6] = {(-1),(-5),(-1)
        ,(-1),(-5),(-1)};
    int32_t l_4 = 0x7E4FC228;
    int16_t l_8 = 8;
    int i;
    for (l_4 = (-1000); (l_4 <= 5);
        l_4 += 1)
    {
        int i;
        l_2[l_4] = 0x5CCB9725;
        l_2[4] = l_2[l_4];
        l_8 = l_2[1];
        l_2[l_4] = l_2[0];
    }
    return l_2[1];
}
```

Fig. 4 Example of out-of-bounds access program generated by CSmith

```
/* CWE193_wchar_t_declare_cpy_68a.c
 */
void badSink();
wchar_t * badData;
void
CWE193_wchar_t_declare_cpy_68_bad()
{
    wchar_t * data;
    wchar_t dataBadBuffer[10];
    data = dataBadBuffer;
    data[0] = L'\0'; /* null
        terminate */
    badData = data;
    badSink();
}

/* CWE193_wchar_t_declare_cpy_68b.c
 */
#define SRC_STRING L"AAAAAAAAAA"
extern wchar_t * badData;
void badSink() {
    wchar_t * data = badData;
    wchar_t source[10+1] =
        SRC_STRING;
    wcscpy(data, source);
    printWLine(data);
}
```

Fig. 5 Example of vulnerable program in Juliet Test Suite

presents a challenge for analysis tools, as not only do they have to consider the statements and instructions within a function itself when trying to identify errors, but also all callers of that function when the parameters are passed by reference. In this case, the caller is in `68a.c`, and the sink in `68b.c`), which means that an analysis tool which only examines one file in isolation at once would be unable to detect these errors, except for generating a general warning about using potentially unsafe functions such as `wcscpy`, instead of its bounds checked alternative, `wcsncpy`.

4.2.1 Test case grouping

While the Juliet Test Suite does contain build files, it was insufficient for our purposes since it packages all test cases together in a single binary, and we need to be able to label each test case with the weakness contained within. As mentioned in Sect. 4.2, test cases can span multiple source files; therefore, we developed a pre-processor which groups the source files for each test case.

The filenames of the test cases follow a predictable pattern; therefore, it is possible to group each

case's files together with simple string manipulation. Each of the files usually has two sections, one guarded by a pre-processor check for `OMITGOOD`, and another guarded by `OMITBAD`. We use the inverse of these checks to generate the labeled samples, i.e., `OMITGOOD` becomes the "vulnerable" sample, and `OMITBAD` becomes the "non-vulnerable" sample. To remove the sections of the program not applicable to the label, the macro is defined at compile time, when the LLVM bitcode is produced (see Sect. 5.1), prior to VFG generation.

Vulnerable samples were labeled with the CVE number the test case is associated with, for example, `CWE121`. Non-vulnerable samples across all weakness categories are grouped together under the label `GOOD`, regardless of which weakness they were demonstrating.

4.2.2 Label selection

One of the issues with the Juliet Test Suite is the large range of test case counts across weakness classes. Some classes have thousands of test cases; whereas, others only have tens or hundreds. Such small numbers are insufficient for training neural networks; therefore, we chose a subset of memory and overflow-related errors that we believed would be the best fit for value flow analysis, rather than using the full dataset. Table 1 shows the classes we selected for our evaluation, with the number of samples evenly split between vulnerable and non-vulnerable for each class.

Table 1 Subset of Juliet Test suite utilized

Group	Description	Count
CWE78	OS command injection	3040
CWE121	Stack-based buffer overflow	8072
CWE122	Heap-based buffer overflow	5008
CWE124	Buffer underwrite	2576
CWE126	Buffer over-read	1944
CWE127	Buffer underread	2576
CWE134	Uncontrolled format string	4180
CWE190	Integer overflow	6840
CWE191	Integer underflow	5244
CWE194	Unexpected sign extension	1824
CWE195	Signed-to-unsigned conversion	1824
CWE197	Numeric truncation error	1368
CWE369	Divide by zero	1368
CWE401	Memory leak	1736
CWE590	Free memory not on heap	1224
CWE690	NULL Deref from return	1444

5 Data pre-processing

This section describes our method for pre-processing programs, producing the inputs to the classification model. The same pre-processing method would be followed for utilizing the trained model to make predictions in a practical application.

5.1 Program compilation

The SVF framework we utilize to produce the inter-procedural value flow graphs operates on LLVM modules; therefore, the C programs in our datasets must be compiled to LLVM bitcode to be analyzed. Many of the samples contained data flows via pointers which span multiple translation units (source files), and thus span LLVM modules. (see Sect. 4.2.1). Analyzing each of these modules independently would fail to capture these data flows. To ensure these data flows were captured, we utilized the wllvm (Whole Program LLVM, Ravitch [33]) compiler driver, which retains a copy of the IR representation generated by the compiler frontend for each module. After all modules have been compiled, the *extract-bc* utility is utilized to combine and generate a single bitcode file from one or more original translation units. For the case where multiple translation units have duplicate functions with the same name, `wllvm` will add a suffix to avoid conflicts. Compiler optimization was not enabled during our testing, as many of the programs contain undefined behavior which could be optimized out entirely, leading to mis-predictions. The result of the compilation step was a single bitcode file for each sample.

5.2 IVFG computation

The Whole Program Analysis (WPA) tool from the SVF framework for program analysis [19] was utilized to compute the inter-procedural value flow graph (IVFG) from the LLVM IR (bitcode) modules produced by the compiler. To compute the points-to set required for value flow analysis, Andersen's pointer analysis [23] was utilized. This enables the resulting value flow graph to be alias-aware, and capable of identifying data flows through pointers which would otherwise not be possible by simply examining the data flow graph, as described in Sect. 3.2.

Generation of the value flow graph for each LLVM module was performed by a dataset-specific script, which annotated each graph with the associated label (vulnerability class or `GOOD`). Each of these graphs was written in graphviz (dot) format, suitable for loading by the node embedding process. As the value flow graph produced by the `WPA` tool does not always include the original

instructions which lead to the creation of the VFG nodes, we modified `WPA` to compute the instruction embedding at the time the graph is written to file, and to append this vector to the node attributes in `graphviz` format. This was achieved by loading the pre-computed vectors for each token (see Sect. 5.3), and generating/emitting the instruction embedding at the same time as processing the other per-node attributes.

5.3 Instruction embedding

While the value flow graph provides some information about the original instructions responsible for producing the data flow, some context is lost. For example, a *LoadVFGNode* represents a data load from a pointer, but the type of data are not retained. To provide the model with a greater amount of information about the operations represented by the graph nodes, we compute an embedding based on the operation and operands of the instruction. Computing embeddings from LLVM IR instructions has been used with success in other applications, such as IR2Vec [34], however, not individual instructions in isolation. First, we split the instruction into tokens for its operation (opcode), type, and parameters (operands), as shown in Fig. 6.

Value identifiers (represented by the percentage sign and a number in the original IR) are dropped as they are meaningless outside of the function, when only examining a single instruction. The opcode is preserved based on its mnemonic (`load`). Types for both the parameter and instruction result are generalized; with a single token representing all integer types, another to represent floating point types, pointer types, and so on. Generalizing the types ensures the distance between resulting vectors of instructions with similar semantics is minimized.

Each of these tokens is looked up in an embedding dictionary containing 8-element vectors. A TransE [35] model was trained on triplets formed from the instruction's tokens (opcode, resulting type, and operands), with the relation between these tokens being represented in vector space. The entire Juliet Test Suite was used as a source for generating the triplets, resulting in 48 unique tokens for the IR instructions and types contained within. We wish to retain as much information in our embeddings, for the model to have a greater chance at learning the semantics of the input programs. Therefore, instead of summing the vectors for each token together after applying a weight to each, we concatenate these vectors, producing a longer vector, but with dedicated elements for each component of the instruction.

```
%3 = load i32, i32* %2, align 4
type: i32
opcode: load
parameter: i32*
```

Fig. 6 Splitting LLVM IR Instruction

Examination of the IR produced by compiling the Juliet Test Suite dataset found that the most common number of operands per instruction in SSA form was 2, excluding the Get Element Pointer (GEP) instruction. We chose not to consider the GEP instruction because it is less useful for the instruction/node embedding, since pointers are captured in the graph itself. Therefore, we chose to only include the first two operands in the instruction embedding; any additional operands are ignored, and the corresponding sub-vectors for instructions with fewer than two operands (e.g., loads/stores) are filled with zero. The sub-vectors for operands which are present are populated only with the embedding for the type of the operand, as the SSA identifier (e.g., `%2`) does not provide any information about the data it represents outside of the context of the basic block, and the input relation to the instruction is already captured in the graph itself.

Using the load instruction example from Fig. 7, `load i32, i32* %2, align 4`, this would result in the first 24 elements of the vector being populated, and the last 8 being zeroed. An arithmetic operation, such as `%7 = add nsw i32 %5, %6` would populate all 32 elements.

5.4 Node embedding

The gated graph layer used in our model requires a feature vector to be computed for each graph node, prior to feeding the data into the network. We include a vector representation of the graph node type, enabling the network to make decisions based on the type of data flow (for example, an addition of two variables, or a memory load). The type information captured from the LLVM IR instruction which the node was created from is also included. Combined, this produces a 40-element vector for each of the value flow graph nodes. This 40-element vector contains two sub-vectors:

Node type sub-vector To embed the node type information, we examine all graphs produced by the dataset, and identify the unique node types. For our benchmarks, there was 19 unique node types, such as *LoadVFGNode* or *GepVFGNode*. A Word2Vec [36] model was trained

Fig. 7 LLVM IR Embedding showing an instruction broken into its four components

to produce a vector encoding of these node types, using the Gensim implementation. Each vector is 8 elements wide, with the hyper-parameters for the Word2Vec model set to the defaults.

Instruction sub-vector The LLVM IR instruction which the IVFG node was derived from forms the majority of vector elements. Both the instruction and types of its operands are utilized in the embedding process, described in Sect. 5.3. The node types which are not derived directly from IR instructions are zero-filled, but these represent only a small fraction of the graph.

6 Model

In this section, we describe our proposed neural network model, for both single and multi-class classification of vulnerable programs. Similar network designs were used for a range of applications in [31, 37, 38]. Figure 8 shows a summary of the layers of the model, described in further detail below.

6.1 Input

The input to the model is formed by both the nodes in the sample's inter-procedural value flow graph, and the edges between them representing transfer of data. Each node has a corresponding feature vector, computed based on the IVFG node type, as well as the IR instruction which it was originally produced from (Sect. 5.4).

The implementation of the Gated Graph Sequence layer that we utilized requires the number of nodes in the graph be constant; therefore, we chose a fixed value based on the maximum IVFG size produced for our datasets. For our experiments, this was 1400 nodes, sufficient to fit all the graphs in the dataset, with no upper bound on the number of edges between nodes. Graphs with fewer nodes were padded with zero to match the fixed size.

6.2 Gated graph sequence layer

The Gated Graph Sequence Layer [30] is utilized to produce feature maps based on both the original vector for each graph node, as well as the edges between them. The GGS uses message passing and the Gated Recurrent Unit to reduce the variable number of graph edges to a fixed size output, which can be utilized by the downstream layers and transformations in the model.

In our experiments, we set the output size, or number of channels, to `200`, therefore producing a final feature map of `[1400 x 200]` for each input sample. The number of layers, or rounds performed by the GGS, was set to `6`, with the source-to-target messages, or aggregation summed.

Considering the input node embeddings I_i (a matrix of $EMB\_SIZE \times NUM\_NODES$, batch i), adjacency list A_i, and W as the learned weight matrix, *Propagate* as the message passing function, and *GRU* as Gated Recurrent Unit [39], the Gated Graph Sequence Layer [30] is defined in Torch-Geometric [40] as:

$$M_1 = I_i W \tag{1}$$

$$M_2 = Propagate(A_i, M_1) \tag{2}$$

$$G_i = GRU(M_2) \tag{3}$$

6.3 Convolution layers

While the GGS layer does consider the spatial relation and flows between graph nodes, it still produces a fixed size feature vector for each node in the graph, leading to a very large feature map. To reduce the dimensionality of this feature map, and concentrate on nodes relevant to the classification task, as well as assist with learning relations

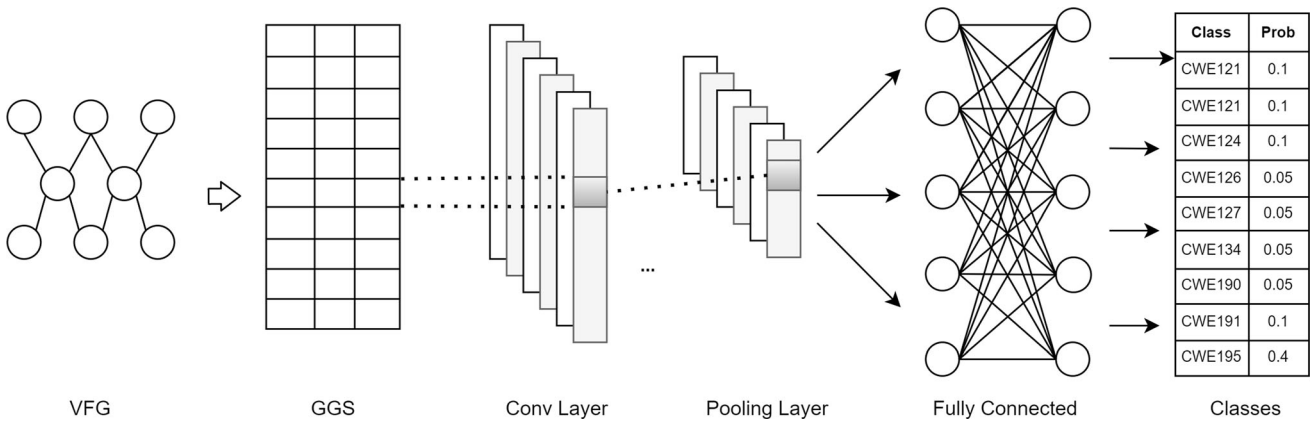


Fig. 8 Visualization of the six stages of our model

between the feature vectors for nearby graph nodes, we utilize three one-dimensional convolutions, each followed by a Max Pooling layer. The number of channels for each of the convolutional layers was set to `600`, `300`, and `150`, respectively, with a kernel size of `6`, `1`, `1`, and padding of `1`. The pooling layers used a kernel size of `3`, `1`, `1`, and a stride of `1`. Both Average and Max Pooling layers were evaluated, with Max Pooling achieving higher performance. After the pooling layers, the rectified linear unit (*ReLU*) activation function is applied, as we found it to provide the most consistent performance compared to sigmoid and hyperbolic tangent functions. After applying the three convolution and pooling layers, we are left with a `[150 x 31]` feature map, or `4650` elements, reduced from the `[1400 x 200]` output of the GGS.

Formally, we define the convolutional operation as $C(x)$, incorporating the single-dimensional convolution, followed by max pooling and the ReLU activation function:

$$C(x) = ReLU(MaxPool(Conv1D(x))) \tag{4}$$

Then, concatenate both the feature map computed by the Gated Graph Sequence Layer G_i with the original node embeddings I_i, producing L_1, then apply the three convolutions in sequence, producing N_i:

$$L_1 = Concatenate(G_i, I_i) \tag{5}$$

$$L_2 = C(L_1) \tag{6}$$

$$L_3 = C(L_2) \tag{7}$$

$$N_i = C(L_3) \tag{8}$$

6.4 Fully connected layers

The final component of the model is the multilayer perceptron (MLP). The feature map from the convolution layers is flattened to a single dimension, leaving `4650` elements, which is fed through two hidden layers to produce the network's output. In our experiments, we set the dimension of the hidden layers to `1000`, with the final layer size and activation depending on whether the model was being used for single or multi-class prediction. The MLP is responsible for learning the relationship between the large feature maps produced by the GGS and convolution layers, and the much smaller number of output classes that the model predicts. We utilized layer normalization and dropout layers before and after the hidden layers within the MLP, to improve the model's generalization and reduce overfitting on the training set.

Formally, using the output of the convolutional layers N_i, the Layer Normalization function *LN*, Dropout function *Drop*, the linear transformation *Linear*, and the prediction for the batch X_i, this overall layer can be expressed as:

$$F_1 = ReLU(Linear(Drop(LN(N_i)))) \tag{9}$$

$$F_2 = ReLU(F_1 + Linear(ReLU(MLP(F_1)))) \tag{10}$$

$$O_i = Linear(Drop(LN(F_2))) \tag{11}$$

6.5 Model output

For binary classification, we apply the Sigmoid activation function to the output of the fully connected layer, resulting in a value between 0 and 1, representing the probability that the input program contains a vulnerability.

$$X_i = Sigmoid(O_i) \tag{12}$$

For multi-class classification, the last fully connected layer has a dimension of the number of classes plus one, with the additional neuron representing the probability that the input program contains no weaknesses. The softmax activation function is applied to the final output values, producing values between 0 and 1 for each class. The final prediction for the input is selected based on the class that has the highest probability.

$$X_i = Softmax(O_i) \tag{13}$$

7 Evaluation

Experiments were conducted to answer four research questions:

- *RQ1* How effective is our model at identifying samples containing any vulnerability?
- *RQ2* How effective is our model at identifying the type of vulnerability contained in the sample (i.e., which weakness does it contain)?
- *RQ3* How does our model compare to traditional static analysis tools at identifying vulnerabilities in the dataset?
- *RQ4* How does our model compare to other state-of-the-art machine learning-based vulnerability detection systems?

7.1 Experimental setup

To evaluate the performance of our model, we implemented the design in PyTorch [41], using Torch-Geometric [40] for data loading and the Gated Graph Sequence layer. Training and inference was performed on a workstation with an AMD Ryzen Threadripper 3970X CPU, 256GB of DDR4 memory, and an RTX 3090 GPU.

Metrics were computed by training the model on the training set, and evaluating on the test set (unseen

samples). Across both datasets, we considered two scenarios: Multi-Class Prediction (predicting the correct weakness/class for the sample), and Vulnerable Sample Identification (predicting whether the sample contains *any* vulnerability).

We treat the vulnerability detection goal as an information retrieval problem, aiming to identify the vulnerable samples, ordered by the confidence that a vulnerability is contained within. To benchmark performance, we define the following metrics, where *TP* is true positives, *FP* is false positives, and *FN* is false negatives:

1. *Precision* The ability of the model to identify true positives, and not predict false positives (i.e., predicting not vulnerable as vulnerable): $P = \frac{TP}{TP+FP}$
2. *Recall* The ability of the model to identify true positives, without considering false positives (i.e., what percentage of vulnerable programs were identified): $R = \frac{TP}{TP+FN}$
3. *F1* The harmonic mean of precision and recall, giving an indication of overall model performance: $F1 = 2\frac{P \times R}{P+R}$

The dataset generation followed the pre-processing method described in Sect. 5. We performed training and evaluation separately on the two datasets introduced in Sect. 4. Each dataset was split into a training, test and validation set using 80, 10, 10% of the samples, respectively. Each subset was stratified based on the weakness classes to ensure that each class contained approximately the same fraction of the total samples across the splits.

The model was trained for up to `100` epochs, with early stopping being utilized to reduce overfitting on the training set. The validation set was used as the trigger for early stopping; after the validation loss increased for 10 consecutive epochs, training was halted, with the parameters before the first higher-validation-loss epoch restored. Table 2 shows the number of samples in each of the subsets.

For training, the *Adam* optimizer was utilized, with the learning rate set to `1e-4`, a weight decay (L2 penalty) of `1.3e-6`, and a batch size of 64. For the multi-class models, the Cross-Entropy loss function was used, and for the binary classification models, we utilized Binary Cross-Entropy. Other hyper-parameters were left at defaults, with the aforementioned values and layer parameters found to be optimal for the model through binary search.

7.2 RQ1: vulnerable sample identification

Methodology For this experiment, we collapsed all the vulnerable classes into a single, "vulnerable", label. Thus, the model can output one of two predictions, "vulnerable" or "non-vulnerable". While the focus of this article is on multi-class prediction, for comparing our method to prior work, we also required a binary classification model. A separate model was trained and evaluated for both of the datasets, following the procedure discussed in Sect. 7.1.

Results Table 3 shows the precision, recall, and F1 score of the model on the test split of the datasets (samples unseen when training).

The model achieves very high performance on the CSmith dataset, identifying 98.24% of the vulnerable samples. False positive rates were low, with a precision of 96.35% indicating that on average, only 3 out of every 100 programs were falsely reported as vulnerable when they contained no errors.

Discussion Performance on the Juliet Test Suite was slightly lower, where 90.14% of the vulnerable programs were identified. These results were somewhat expected, as the samples in the CSmith dataset (`A`) only represent a single type of vulnerability (out of bounds memory access), in contrast to the Juliet Test Suite (`B`), where we selected a range of vulnerabilities, including integer overflows, and other undefined behavior, which can have significantly different value flows compared to out-of-bounds memory access. However, despite the more difficult problem, the model still achieved low false positive rates, with on average, 9 out of 100 samples without errors being falsely reported as vulnerable.

These results suggest that our model can both identify simple errors matching the out-of-bounds memory access pattern, and in addition, generalize to other types of errors/vulnerabilities, with the caveat that it must be trained with examples of these errors to recognize them.

Table 2 Sample count in each dataset split

Dataset	# Train	# Test	# Validation
Fully synthetic	81600	10200	10200
Juliet test suite	40214	5026	5028

Table 3 Precision, recall and F1 scores for binary classification

Dataset	Precision (%)	Recall (%)	F1 (%)
Fully synthetic	96.35	98.24	97.28
Juliet test suite	90.14	93.35	91.72

7.3 RQ2: multi-class prediction

Methodology This experiment uses the model to predict which weakness class a particular sample falls into, i.e., which vulnerability it contains. We evaluated performance for both the single-weakness-per-model and multi-class models. Multi-class classification was only applicable to the Juliet Test Suite dataset (B), as the fully synthetic dataset does not contain any non-memory-related vulnerabilities, and therefore only spans two classes.

Results Table 4 shows the precision, recall, and F1 scores for using multiple models, one per vulnerability class *SC*, trained separately. *MC* shows the metrics for a single model, trained to recognize all vulnerability classes. Figure 9 shows the same results in graphical form, to easily identify which classes are under-performing.

For multi-class prediction of the Juliet Test Suite Dataset, precision varied, with some classes as low as 45% (CWE191), but others having no false positives (CWE78). We believe two main factors are responsible for this level of performance; the similarities between classes (e.g., multiple integer over/underflow-related errors, or buffer-related errors), and the imbalance of samples across the classes (e.g., CWE197 only has 127 samples, with 62 of those positive; whereas, CWE134 has 454, with 227 positive). This is also reflected in the recall numbers, with a non-trivial difference between classes.

Discussion Comparing the single-class model (one model trained for each class), there is a measurable performance improvement across the board; performance significantly increases across the classes. However, the model-per-class method is not without downsides; for it to provide the same exact-weakness functionality to the user as the multi-class model, one would have to perform a number of predictions equal to the number of classes, and then take the class with the highest probability. These results suggest that the model is capable of successfully identifying which features contribute to the sample being vulnerable versus non-vulnerable, and increasing the number of parameters in the multi-class model may improve performance, as the single-class model effectively has more parameters per class. However, we leave this for future work.

Using a 80% F1 score as a baseline for low performance, we can observe 4 classes that under-perform in the multi-class model: CWE190 (Integer Overflow), CWE191 (Integer Underflow), CWE121 (Stack-Based Buffer Overflow), CWE122 (Heap-Based Buffer Overflow), CWE124 (Buffer Underwrite), CWE126 (Buffer Overread), and CWE127 (Buffer Underread).

CWE190 and CWE191 are similar, overflow vs. underflow, with the only difference being the instruction opcode. However, subtraction can also be expressed as the addition of a negative number. LLVM IR also does not differentiate between signed and unsigned types, and model may also struggle to predict which of these two classes the sample falls within, as the higher-level semantics are lost during the compilation process. The fact that the single-model-per-class configuration (MC) achieves 97-98% performance supports this hypothesis since it does not need to decide whether it is an underflow or overflow specifically, just one of the two.

CWE121, CWE122, CWE124, CWE126, and CWE127 are all memory buffer-related errors, with the common property of out-of-bounds access. The CWE categorization presents a challenge to classifying these samples, as the

Table 4 Precision, recall and F1 scores for single vs. multi-class models

Class	#	MC Prec	MC Recall (%)	MC F1 (%)	SC Prec (%)	SC Recall (%)	SC F1(%)
CWE121	882	75.97	74.75	75.36	80.16	74.01	76.96
CWE122	486	69.96	70.66	70.31	74.49	72.40	73.43
CWE124	240	74.90	75.19	75.05	82.71	85.27	83.97
CWE126	208	67.51	68.21	67.86	73.04	86.60	79.25
CWE127	274	76.30	79.84	78.03	78.77	89.15	83.64
CWE134	454	95.56	97.85	96.69	93.09	96.65	94.84
CWE190	670	56.53	56.29	56.41	97.96	98.82	98.39
CWE191	460	45.09	47.33	46.18	98.11	98.85	98.48
CWE194	174	88.24	90.66	89.43	96.43	89.01	92.57
CWE195	170	86.89	87.36	87.12	98.78	89.01	93.64
CWE197	124	82.82	98.54	90.00	100.00	100.00	100.00
CWE369	152	94.12	81.75	87.50	86.67	95.59	90.91
CWE401	172	96.73	85.06	90.52	97.70	97.70	97.70
CWE590	134	94.07	90.98	92.50	98.33	96.72	97.52
CWE690	138	93.70	82.64	87.82	89.55	83.33	86.33
CWE78	288	100.00	99.34	99.67	99.35	100.00	99.67

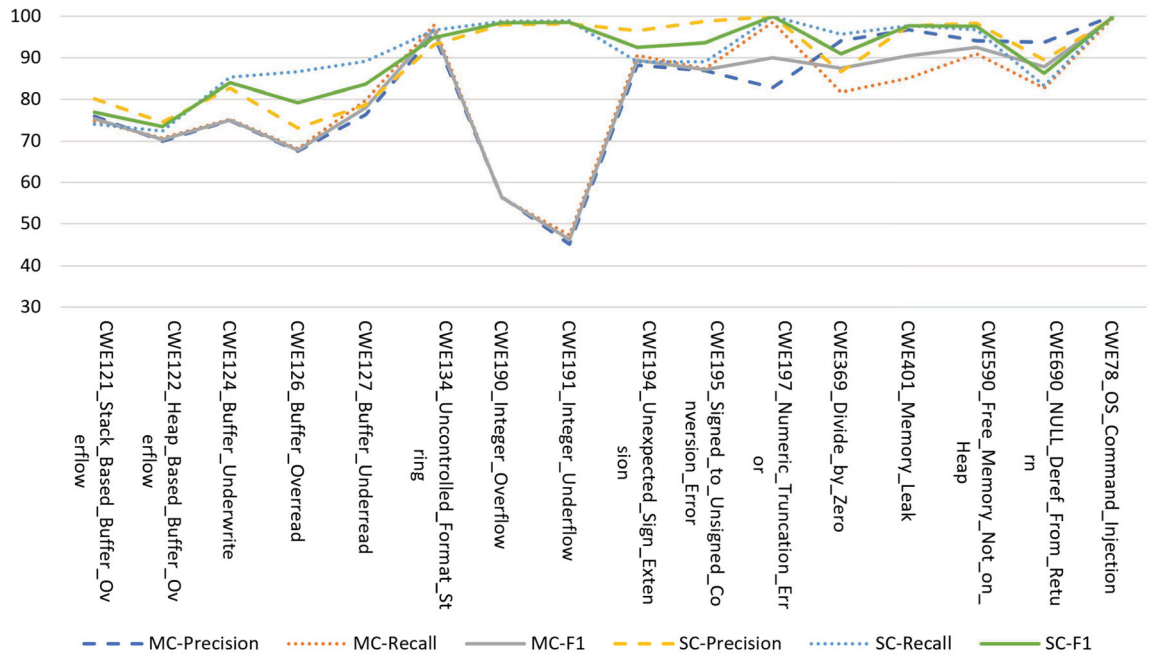


Fig. 9 Precision, recall and F1 scores for single vs. multi-class models

vulnerability may be a stack-based buffer overflow (CWE121), but also a buffer over-read (CWE126). This is in contrast to other machine learning applications, where classes are distinct with minimal overlap. Therefore, a model that reports any of these classes in the top predictions may arguably be correct, but due to being constrained by a single label, will not count as a true positive.

To gain insight into how much of a performance impact the multi-class model has over the binary classification model (Sect. 7.2) at identifying *any* vulnerability, we compared the precision, recall and F1 scores for both models, after collapsing the classes of the multi-class model to a binary result, where any of the 16 vulnerability classes was treated as "vulnerable". Table 5 compares these results for Dataset B (Juliet Test Suite).

Despite the lower performance in some of the vulnerability classes of the multi-class model, it still potentially provides value for users. Comparing the multi-class model, with the predictions binned as vulnerable or non-vulnerable, to the single-class model (Table 5) shows that the multi-class model achieves similar performance to a binary model, with only a difference of 1.27% in precision, 1.76% in recall, and 0.22% F1. This indicates that while the multi-class model may predict the incorrect weakness class for a program, it is still able to identify which programs are vulnerable versus non-vulnerable with high precision (low false positive rates), on average 9 out of 10 times.

Table 5 Precision, recall and F1 scores for multi-class vs. binary model for vulnerability detection

Model	Count	Precision (%)	Recall (%)	F1 (%)
Multi-class	5026	91.41	91.59	91.50
Single-class	5026	90.14	93.35	91.72

7.4 RQ3: static analysis tools comparison

Methodology To compare our work with more traditional program analysis methods for bug detection, we utilized two popular static analysis tools: cppcheck [42], version `2.3`, and the Clang static analyzer, part of the LLVM compiler framework [13, 43], version *12.0.0*. These static analysis tools can identify a range of potential issues in programs, including bugs which lead to security vulnerabilities.

The performance of these tools was evaluated using the same information retrieval metrics that we used to determine our model's performance (precision, recall and F1 score). We chose to evaluate the Juliet Test Suite (Dataset B), rather than the CSmith dataset, as these tools are known for being able to detect many types of vulnerability causing errors, not just memory/buffer-related issues. To generate the inputs for the tools, the same grouping method described in Sect. 4.2.1 was followed, generating two samples for each test case. All the source files for each of the programs were provided to the analyzer, alongside the utility file, containing additional functions and declarations (e.g., printing output to the terminal). We treated any

output from the analyzer, both warnings and errors, as a vulnerable report, with no output being considered non-vulnerable.

Results Table 6 shows the precision, recall and F1 scores of our model, versus both the cppcheck and clang static analysis tools. The performance of the Devign [31] vulnerability detection system is included in these results, and discussed in Sect. 7.5.

Discussion We can observe that our graph-based model significantly outperforms the static analysis tools across all three metrics. The lower precision suggests that the static analysis tools produced a high number of false positives, which we confirmed through manual examination of the outputs; many of the samples were reporting the same warnings for both the vulnerable and non-vulnerable variants of the same program. Recall was also much lower, suggesting a high false negative rate. Between the two static analyzers, Cppcheck appears to produce fewer false positives, at the cost of missing more errors than the Clang static analyzer. However, as aforementioned, we treated any output from the tools as a "vulnerable" label; therefore, the higher recall in the Clang results can be explained by the same false error report in both the vulnerable and non-vulnerable samples in some cases.

7.5 RQ4: How does our model compare to other state-of-the-art machine learning-based vulnerability detection systems?

Methodology To discover how our model performs relative to recently published, machine learning-based vulnerability detection systems, we utilized the Juliet Dataset (4.2), comparing the prior work to our model using the three metrics introduced in Section 7: Precision, Recall and F1 score.

We were able to evaluate our dataset on one of the open-source implementations [44] of the Devign [31] paper. The Juliet dataset was split to the same 80/10%/10% arrangement for training/testing/validation samples, with the training parameters set similar to our model: 100 epochs with early stopping enabled.

Table 6 Precision, recall and F1 scores of our model vs. other detection systems

Model	Precision (%)	Recall (%)	F1 (%)
Cppcheck	59.69	9.74	16.75
Clang	41.06	27.64	33.04
Devign	72.13	65.59	68.71
Our model	90.14	93.35	91.72

In addition, we originally planned to compare the performance of our model with VulDeePecker [9] and SySeVR [45]. However, we found that with the implementation released in [46], the call graph generation step did not complete after several weeks of runtime. Several issues in the GitHub repository report similar difficulties [46]. Furthermore, other more recent publications did not have public implementations available at the time of submission. Due to these publications utilizing different datasets, we could not directly compare their performance numbers on the same dataset we used for evaluating our model.

However, as the dataset for the VulDeePecker paper has been released, and since it was based on the same Juliet Test Suite samples, we are able to compare how our model performs relative to VulDeePecker on the subset of classes that Li et al. evaluated [9]. These two evaluations are reported as CWE-119 and CWE-399; however, these classes include several CWE groups (e.g., CWE-119 includes CWE-121, CWE-122, etc.). We evaluated our model in both multi-class and single-class modes, to determine its effectiveness at differentiating between similar, but different vulnerability classes, something which is not possible in VulDeePecker.

Results

Table 6 compares the precision, recall and F1 scores of our model, compared to Devign [31, 44] on the same dataset. As the Devign model only considers vulnerable versus non-vulnerable labels, we compared it to the results of our binary classification model (Sect. 7.2). Compared to Devign, our model achieves 25% greater precision, 39% greater recall, and a 31% greater F1 score on the Juliet dataset. Both machine learning-based models outperform the static analysis methods introduced in the previous section by a considerable margin.

Table 7 and Figure 10 show the precision, recall and F1 scores for our model *(VFG)* in single-class mode (7.2) versus VulDeePecker *(VDP)* [9]. We also benchmarked the

Table 7 Performance comparison on VulDeePecker dataset

Model	Dataset	Precision (%)	Recall (%)	F1 (%)
VDP	CWE-119	91.7	82.0	86.6
Clang	CWE-119	46.5	30.0	33.3
Cppcheck	CWE-119	56.4	13.8	22.2
VFG	CWE-119	85.3	83.2	84.3
VDP	CWE-399	94.6	95.3	95.0
Clang	CWE-399	57.8	56.6	57.2
Cppcheck	CWE-399	69.4	22.4	33.9
VFG	CWE-399	94.4	96.6	95.5

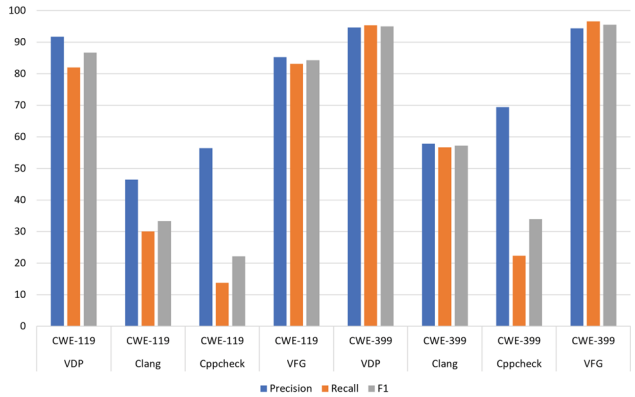


Fig. 10 Performance comparison on VulDeePecker dataset

two previously introduced static analysis tools, Clang and Cppcheck on the VulDeePecker dataset. As aforementioned, the datasets do not consist of a single CWE label from the Juliet Test suite, a full breakdown of the included classes is shown in Table 8.

Compared to VulDeePecker [9], our model achieves similar performance, sacrificing a small amount of false positives for slightly higher true positive detection. As VulDeePecker cannot predict different labels for an input, only a binary vulnerable/non-vulnerable label, we cannot compare our model's ability to differentiate between different weakness classes. For completeness, we included the results for multi-class prediction from our model in Table 8.

Table 8 Multi-class precision, recall, and F1 on VulDeePecker dataset

Dataset	Precision (%)	Recall (%)	F1 (%)
CWE121 [CWE-119]	78.23	72.03	75.00
CWE122 [CWE-119]	70.85	76.80	73.70
CWE123 [CWE-119]	100.00	90.91	95.24
CWE124 [CWE-119]	79.46	68.99	73.86
CWE126 [CWE-119]	74.65	54.64	63.10
CWE127 [CWE-119]	77.42	74.42	75.89
CWE134 [CWE-119]	97.04	94.26	95.63
CWE415 [CWE-119]	100.00	86.96	93.02
CWE416 [CWE-119]	100.00	71.43	83.33
CWE680 [CWE-119]	100.00	52.17	68.57
CWE401 [CWE-399]	95.29	93.10	94.19
CWE416 [CWE-399]	100.00	92.86	96.30
CWE590 [CWE-399]	93.75	98.36	96.00
CWE761 [CWE-399]	93.18	91.11	92.13
CWE773 [CWE-399]	62.50	62.50	62.50
CWE775 [CWE-399]	100.00	87.50	93.33
CWE789 [CWE-399]	100.00	89.47	94.44

A performance difference can be observed in Table 8 when compared with our earlier evaluation in Sect. 7.3. We hypothesize this is due to the fact that these are two separate datasets being trained and evaluated in isolation, rather than a single model being utilized for all classes.

Discussion

The similar performance of our model when compared with VulDeePecker [9] suggests that graph representations of programs can be just as effective for security vulnerability detection as source-based representations, including slice or gadget-based such as those used by VulDeePecker. However, it is also clear that the chosen graph representation is key to high performance, with the Devign implementation utilized [44] relying exclusively on abstract syntax trees performing significantly lower. Both machine learning vulnerability detection systems outperformed the static analysis tools by a considerable margin, with Cppcheck trailing Clang, similar to the experiment in Sect. 6, although the gap was smaller on the VulDeePecker dataset, particularly in CWE-399.

The authors of the Devign implementation note that not the all code representations mentioned in the original paper [31] were utilized, and that they were unable to reproduce the original results of the paper due to dataset mismatches, among other issues [44]. Therefore, we are unable to reason about how much performance could be improved by including control and data flow information as the original paper proposed. Given the performance discrepancy between our model and Devign, it would suggest that utilizing only representations lacking flow information is insufficient for achieving high detection rates.

Regardless, even with the disadvantages of the open-source implementation when compared to the original paper, it still provides some indication that our model can compete with, or in some cases, outperform, recent state-of-the-art vulnerability detection systems, while introducing the ability to differentiate between different weakness classes.

8 Ablation study

To better reason about the performance of our model, we propose two further questions:

1. *Feature Impact* Which of the additional features that we add to the data has the greatest impact on model performance, and how does the model perform when making predictions solely on the graph itself, without attributes (node types, instruction vectors).
2. *Dataset subsampling* How large does sampling different portions of the dataset affect the performance of the model?

3. *Program representations* How much of the model's performance can be attributed to the use of graph neural networks, versus value flow and instruction embedding?

8.1 Feature impact

To determine the performance impact on the model for the different features we generate during pre-processing, another series of experiments were performed with each of these features disabled.

- *Full Node Embedding (F)* All features were enabled in the model, as described in Sect. 5.4.
- *No Node Types (T)* The node type sub-vector (as described in Sect. 5.4) is set to a fixed vector, leaving only the instruction vector variable.
- *No Instruction Vector (I)* The instruction embedding sub-vector (as described in Sect. 5.3) is zeroed, leaving only the node type. This effectively makes all nodes behave the same as nodes not derived from LLVM instructions.
- *No Node Embedding (N)* All pre-processing features were disabled, by setting the entirety of the node's attribute vector to zero.

The same 16 weakness classes evaluated in Sect. 7.3 were examined, recording the performance for each of the configurations listed above. For all four configurations, the model was trained for 5 epochs, and early stopping was disabled, as we wanted the only variance between runs to be the feature vectors in the node attributes. Table 9 shows the precision, recall, and F1 scores of each configuration.

The extreme outlier in these results is the "No Node Embedding" (N) columns, where the model's performance completely collapses, where the model's recall dropped 1% for some classes (e.g., CWE-126). We hypothesize that this is due to the embedding being all-zeros, the model cannot differentiate between present nodes in the graph and the all-zero nodes which are added as padding to fit the fixed input size, and the model is effectively predicting randomly. Therefore, we can ignore these results in further analysis. Initially, for the "No Node Types" case, we set the node type component (the first 8 elements) of the embedding to zero; however, this resulted in similar results to the all-zeros case, as some node types do not contain an instruction vector. Instead, for the "No Node Types" case, we set the node type component to a fixed 8-element value, so while there is no unique information from the node type component, we do not end up with a large number of non-padding all-zero nodes.

The greatest decrease was observed when disabling the node type vectors ("No Node Types", "T"), relying exclusively on instruction vectors. This resulted in an average reduction of 19.5% precision, 19.3% recall, and 22.8% in F1 score. We hypothesize that this is because every node in the graph will have a corresponding node type vector, whereas as mentioned above, some of the VFG-specific nodes, such as formal in/out parameters, are not directly generated from an instruction, and thus will have an instruction vector of zero.

Disabling the instruction vector ("No Instruction Vector", "I") has a smaller impact, with a 5.6% reduction in recall, and approximately a 1.5% difference in precision/F1 score. This would suggest that while the instruction embedding does not contribute as significantly to the overall performance of the model as the node types, it still improves detection rates (recall), without resulting in a non-trivial change to false positive rates (precision), especially for some classes. For example, CWE-369 (Divide By Zero), where the F1 score decreases by 23.2% without instruction vectors. In this case, it makes sense that the instruction vector provides valuable information for the model, as the impact of a zero parameter on a division operation is much more significant than say, an addition or subtraction operation.

8.2 Dataset subsampling

While many of the samples in the Juliet Test Suite were very similar in how the flaw was represented in the program code, some samples contained significant differences. To examine the impact of the training/testing/validation split on the dataset, we utilized K-fold cross validation, computing precision, recall and F1 for each fold. Model performance was evaluated across 5 folds, stratified based on the weakness label to ensure the distribution of classes across the training and testing sets. The train/test split was set at 80%/20%. The hyper-parameters for the model were identical to the results presented in Sect. 7.2, and early stopping with a patience value of 10 was used. Table 10 shows the results of this experiment.

While the precision did vary by up to 2.2%, the recall was much less noisy, only varying by up to 0.9%. The F1 score reflects this, also indicating low variance. This shows that while the selection of training and testing samples does impact model performance, as we expected due to large differences in the semantics and structure of the programs, the accuracy of the model in identifying true positives was largely unaffected, with only a small percentage of additional false positives reported based on the dataset split.

8.3 Program representations

In Sect. 7.5, we evaluated our model in the binary classification task and compared the performance to the Devign

Table 9 Model performance with subset of input features

Class	F-Prec (%)	F-Rec (%)	F-F1 (%)	T-Prec (%)	T-Rec (%)	T-F1 (%)	I-Prec (%)	I-Rec (%)	I-F1 (%)	N-Prec (%)	N-Rec (%)	N-F1 (%)
CWE121	75.35	72.79	74.05	89.01	32.40	47.51	76.33	65.08	70.26	82.22	16.78	27.87
CWE122	67.75	76.95	72.06	37.61	31.78	34.45	74.35	70.37	72.30	87.50	5.76	10.81
CWE124	81.93	56.67	67.00	52.58	52.58	52.58	78.31	54.17	64.04	25.00	1.67	3.12
CWE126	65.26	59.62	62.31	60.76	37.21	46.15	65.82	50.00	56.83	100.00	0.96	1.90
CWE127	73.91	74.45	74.18	87.56	87.56	87.56	85.71	61.31	71.49	65.22	10.95	18.75
CWE134	98.65	96.92	97.78	45.15	91.23	60.41	98.11	91.63	94.76	52.38	87.22	65.45
CWE190	57.68	82.99	68.05	38.71	9.16	14.81	57.68	63.88	60.62	51.08	77.61	61.61
CWE191	42.22	16.52	23.75	86.84	72.53	79.04	41.95	43.04	42.49	44.90	9.57	15.77
CWE194	95.31	70.11	80.79	78.33	51.65	62.25	92.21	81.61	86.59	83.33	28.74	42.74
CWE195	75.56	80.00	77.71	56.14	46.38	50.79	82.72	78.82	80.72	100.00	7.06	13.19
CWE197	94.64	85.48	89.83	55.32	37.68	44.83	97.67	67.74	80.00	56.67	27.42	36.96
CWE369	86.76	77.63	81.94	83.72	41.38	55.38	81.25	51.32	62.90	81.82	11.84	20.69
CWE401	89.23	67.44	76.82	54.35	81.97	65.36	85.14	73.26	78.75	40.34	55.81	46.83
CWE590	100.00	77.61	87.39	48.06	86.11	61.69	98.11	77.61	86.67	66.18	67.16	66.67
CWE690	86.89	76.81	81.54	83.52	96.71	89.63	94.74	78.26	85.71	90.91	14.49	25.00
CWE78	100.00	100.00	100.00	81.99	88.98	85.34	99.30	97.92	98.60	75.76	69.44	72.46

Table 10 Performance of model across a range of train/test/validation folds

	Train	Test	Precision (%)	Recall (%)	F1 score (%)
#1	40214	10054	91.33	90.29	90.81
#2	40214	10054	92.70	88.60	90.60
#3	40214	10054	92.78	89.24	90.98
#4	40215	10053	89.89	93.27	91.55
#5	40215	10053	89.15	93.35	91.20

[31] paper. These results show a clear improvement in both precision and recall from using the VFG and instruction embedding representation, over Devign, which uses the Abstract Syntax Tree. This highlights the benefits of using value flow graphs. However, it is unclear which contributes the most: the value flow or graph neural networks.

To gain insight into the benefit of graph neural networks, we utilized the same Juliet Test Suite dataset previously introduced in Sect. 4, and developed a bi-directional GRU (Gated Recurrent Unit) model, that processes a stream of tokens generated from the Abstract Syntax Tree of the sample. This is similar to Devign's use of AST and enables us to isolate the graph neural network, determining its impact.

Utilizing the Clang [43] compiler, we wrote a tool that produces a stream of tokens, while walking the AST of the program. To reduce the size of the vocabulary, local variables are renamed to `LVARn`, where `n` is an increasing counter, global variables are renamed to `GVARn`, parameters are renamed to `PARAMn`, non-primitive types are renamed to `TYPE`, and functions outside of the standard C library are renamed `FUNCTION`. Figure 11 shows an example of the corresponding token stream for the sample C function.

We utilized the Gated Recurrent Unit (GRU) [39] instead of LSTM layers due to their higher performance on smaller datasets [47]. The model itself can be defined as the following, with I representing the input, and O representing the output.

$$C_1 = MaxPooling(Conv1D(I)) \tag{14}$$

$$C_2 = Bidirectional(GRU(C_1)) \tag{15}$$

$$C_3 = Bidirectional(GRU(C_2)) \tag{16}$$

```
void func() {
  if (globalVar) {
    int x = (random() % 3);
  }
}
```

```
IF GVAR1 { LVAR1 int LVAR1 =
    FUNCTION ( ) % 3 }
```

Fig. 11 Example token stream for simple C program

$$C_4 = Dropout(Dense(C_3)) \quad (17)$$

$$O = Softmax(Dense(C_4)) \quad (18)$$

The convolution layer set the filters to 32, with a kernel size of 64. The pooling layer also used a size of 32, and the GRU layers were set to 128 units, as was the hidden dense layer. Before being fed into the convolution layer, each of these tokens is embedded into vector space using a Word2vec [36] model. As the number of tokens is variable, and the fully connected layer expects a fixed-size input, we use the feature vector that is output from the final time step (token) as the output of the bi-directional layers, which includes information from previous tokens.

The GRU model was implemented using the Keras framework and optimized using the Adam algorithm, with all other hyper-parameters left as default. The model was trained for 20 epochs, finding no improvement in validation loss after this number. Table 11 shows the result of this experiment, evaluated in the same manner described in Sect. 7.2.

We can observe that the GRU model achieves 15% greater precision over Devign, but the difference in recall is within the margin of error. The F1 score is, therefore, higher on the GRU model by 6.6%. The VFG model improves precision by 8.3% compared to the GRU model, recall by 42%, and F1 score by 25.1%. Therefore, we hypothesize that while graph neural networks can benefit security vulnerability detection applications, the chosen graph representation is key to high performance. Using the AST alone, as in Devign, still misses 34.4% of samples in our dataset, compared to the 6.7% of the VFG model. These results suggest that our choice of graph representation (value flow graphs and instruction embedding) is the main contributor to the model's performance, as the overall accuracy of the GRU and Devign models was similar.

However, despite the greater performance in the binary classification task, does the choice of representation impact the detection performance of some vulnerability types over others? Are some vulnerability types better expressed with high-level representations, such as the AST, or low-level, such as VFG/LLVM IR? To test this hypothesis, we also trained and evaluated the VFG model in multi-class mode, producing per-class metrics in the manner described in Sect. 7. The results of this experiment are shown in Table 12. Devign and VulDeePecker are not multi-class models; thus, we cannot equally compare with these models.

Table 11 Binary classification of VFG compared to AST (Graph and GRU) models

Model	Precision (%)	Recall (%)	F1 Score (%)
VFG (Graph)	90.14	93.35	91.72
AST (Graph/Devign)	72.13	65.59	68.71
AST (GRU)	83.23	65.43	73.27

We can observe that while the GRU model had lower overall precision and recall in the binary classification task, some classes are more precise with the GRU model. For example, CWE190 and CWE191 (Integer Overflow/Underflow). We hypothesize that this is due to the value flow graph and instruction embedding containing insufficient information for the model to reason about whether the operation is an overflow or underflow; whereas, the high-level information from the AST better represents the calculation. Excluding integer overflow across the board, the majority of classes favored the VFG model in overall accuracy (F1 score).

These results support the hypothesis that different types of vulnerabilities are better represented by high or low-level information. In multi-class classification, the GRU model detected a greater number of vulnerable samples in 4 of the 16 evaluated classes. This suggests that the representation of data is key not only to detect the presence of vulnerabilities, but also to enable the model to differentiate between types in a multi-class context.

9 Related work

Examining the methods of vulnerability detection systems used throughout the literature, most fall into three distinct categories: *rule-based*, *similarity-based*, and *machine learning-based* [48].

Rule-based systems attempt to detect errors in programs by iterating through a set of pre-defined specifications, and testing program code against each rule. These rules are often combined with program analysis techniques to produce the necessary information, such as control and data flow graphs, as seen in the clang static analyzer [27]. The specifications that these rules follow are written by experts [48] and are thus often very generic in nature, limiting the ability for these systems to detect domain-specific errors. While rule-based systems have near-ideal detection granularity, often reporting the exact line number where the error is contained, high false positive rates [9, 48] limit their usability in practical scenarios. Examples of rule-based systems include the previously-examined clang static analyzer [27], cppcheck [42], and commercial tools such as Coverity [49].

Similarity-Based Systems were actively researched in the previous decade, although have lost popularity to machine learning-based systems in more recent years. These types of systems usually attempted to identify bugs through the detection of "code clones" [50], two pieces of

Table 12 Multi-class classification of VFG compared to AST (Graph and GRU) models

Class	V-Prec (%)	V-Rec (%)	V-F1 (%)	G-Prec (%)	G-Rec (%)	G-F1 (%)
CWE121	75.97	74.75	75.36	81.32	79.62	80.46
CWE122	69.96	70.66	70.31	63.86	56.63	60.03
CWE124	74.90	75.19	75.05	60.59	77.72	68.10
CWE126	67.51	68.21	67.86	85.59	71.43	77.87
CWE127	76.30	79.84	78.03	100.00	47.28	64.21
CWE134	95.56	97.85	96.69	63.49	48.43	54.95
CWE190	56.53	56.29	56.41	91.37	48.54	63.40
CWE191	45.09	47.33	46.18	88.24	48.78	62.83
CWE194	88.24	90.66	89.43	94.59	53.03	67.96
CWE195	86.89	87.36	87.12	56.91	81.40	66.99
CWE197	82.82	98.54	98.54	81.08	45.45	58.25
CWE369	94.12	81.75	87.50	61.90	61.90	61.90
CWE401	96.73	85.06	90.52	87.50	93.90	90.59
CWE590	94.07	90.98	92.50	80.77	79.25	80.00
CWE690	93.70	82.64	87.82	70.27	81.25	75.36
CWE78	100.00	99.34	99.67	93.28	48.05	63.43

code which perform the same computation, but may have different syntactic structure. Similarity-based systems generally have high precision, as they will only report errors in code which has been labeled as vulnerable, but due to relying on exact or near-exact matching, also have very high false negative rates (missed detections). Examples of such systems include ReDeBug [51], and VUDDY [52].

Machine-Learning Systems utilize models trained on vulnerable datasets to identify vulnerabilities in programs. While they are also a similarity-based system, an additional level of indirection is introduced by the training process, with the models utilizing features computed from the input program to make classifications, rather than attempting to match the program exactly as a code clone. Utilizing machine learning is not a new approach for bug detection, with publications dating back to 2012 using text analysis techniques, combined with a support vector machine for detecting errors [8]. More recently in the literature, neural network-based approaches have dominated; however, it is not a clear-cut solution to the problem, with more traditional machine learning approaches such as random forests still being utilized in recent research [10, 53]. Neural network approaches range from LSTM-based (Long Short-Term Memory) gadget classification of intra-procedural slices [9], as well as convolution approaches based on natural language processing (NLP) techniques [12]. The inputs used by these models include source code [10, 11, 54], abstract syntax trees [55, 56], and assembly/compiled instructions [12]. In addition to learning directly from the program's code, using metrics computed from the source has also been explored [53].

From these systems, the most relevant to our work is VulDeePecker [9], and Devign [31]. VulDeePecker is a machine learning-based vulnerability detection system developed by Li et al. [9], published in 2018. VulDeePecker divides functions into slices, known as "code gadgets", after stripping whitespace and normalizing variable names. The gadgets are then embedded, with an LSTM (Long Short-Term Memory) layer used to produce fixed-length vectors for variable length slices, and fed into a dense layer for classification. The main downside to VulDeePecker compared to our work is the lack of inter-procedural classification: detections are limited to within a single function only, not considering the full context and callees. The same authors went on to publish SySeVR [45], which instead of using a simple token-based representation of code gadgets, grouped related statements together based on control and data flow. SySeVR [45] was later extended to utilize inter-procedural slices, solving the main limitation of VulDeePecker [9].

Devign, by Zhou et al. [31] uses a range of representations of the input program, including the abstract syntax tree, control flow graph, data flow graph, and an embedded form of the original source tokens. The authors proposed the *Conv* module as a method for computing features for nodes based on the graph representations of the program, and our network follows a similar structure. As with VulDeePecker [9], its major weakness is the intra-procedural nature of its analysis and detection, that is, the inability to detect vulnerabilities spanning across function calls.

In contrast to the methods we are proposing, the majority of this prior work has been focused around analysis of the program source code, which can reduce performance, requiring the model to learn and identify control

and data flow patterns which contribute to vulnerabilities. More recent work such the aforementioned Devign [31], directly utilized control and data flow graphs, combined with AST (Abstract Syntax Tree) paths for bug detection, with non-trivial performance improvements. This development heavily influenced our decision to utilize value flow graphs in our method, making the detection system alias-aware, which is particularly important for memory-related errors.

However, we argue that while the AST is a more general representation of the program compared to the source code, the types of nodes in the tree are tied to the language being analyzed. In contrast, using intermediate representations of the program, usually utilized by compilation, such as Control and Data Flow Graphs, is not only less specific to the original language of the program, but have also been shown to improve detection performance [31]. Our work builds upon this knowledge, using a state-of-the-art, alias-aware, inter-procedural program representation. This choice was driven by the fact that many vulnerabilities discovered today are both complex in nature, spanning multiple functions, as well as being dominated by memory errors [17, 18].

In other applications, such as semantic labeling of functions, some success has been found by utilizing the Abstract Syntax Tree from the program, in particular, the paths between nodes in the tree. Code2vec is a neural network model developed by Alon et al. [25] which represents snippets of code (functions in the paper) as "continuous distributed vectors". The goal of the model is to predict semantic labels, or method names from the features of the code. Code2vec used AST paths as an input into the network, in contrast to the prior work which had used tokens with bag of words models. An AST path represents all the intermediate nodes in a tree between two terminals. The authors argue that using the syntactic paths can "capture regularities that reflect common code patterns", "lowers the learning effort" [25], and can scale to large code bases, since the model does not have to learn the syntactic structure of the programming language. Other works have also investigated the use of AST paths for identifying errors with promising performance [55, 56].

Similar to Code2vec, value flow graphs have also been utilized in recent years for semantic labeling and code captioning tasks (Flow2Vec, Sui et al. [24]). While Flow2Vec utilized higher-order proximity embedding to transform the value flow graph to a vector representation, other recent work has explored generating code embeddings from value flow graphs, also for vulnerability detection, but with a focus on binary classification [21, 22], as mentioned in Sect. 3.2. As in many other applications, transformers have also seen use for vulnerability detection in late 2022 [57].

Initially, for this work, we considered using AST paths as a representation for detecting vulnerabilities. However, we ended up switching to value flow graphs for three main reasons. Firstly, as the paths are local to the function, it prevents us from identifying bugs spanning multiple procedure calls. Secondly, operating at the AST level instead of a compiled intermediate representation results in the semantics of the language being captured in the paths, which means we would require a greater number of samples to capture variations of bugs, which may be more similar once compiled. Lastly, most critically, Code2vec's main limitation is in how the paths are computed—the number of intermediate nodes between terminal nodes will vary between paths, unsuitable for the later layers in the neural network. Code2vec solved this by using the hash of the path, followed by an attention layer to select the most relevant paths [25]. Tying into the second point, this would further reduce the number of samples required to train on, as any small semantic change would change the resulting hash. This weakness was partially solved in Code2seq [58], generating the path embeddings with an LSTM model; however, we did not investigate using Code2seq for bug detection. An alternative method using an RNN for working around the limitations of Code2vec was also developed in 2023 [59].

10 Conclusion

We presented a new approach for identifying the weakness class which a program falls into, based on inter-procedural value flow graphs, and graph neural networks. The performance of the classifier was on average high, capable of handling errors spanning multiple translation units, and outperformed industry-leading static analysis tools. Additional experiments were conducted to reason about the model's performance, and identify which types of vulnerabilities our approach can most accurately detect.

While our model did not detect every vulnerable program (i.e., a recall of 100%), we argue that for the problem of vulnerability detection, at least at scale, precision, is more important to the programmer than recall. High false positive rates, i.e., low precision, mean that the programmer must spend large amounts of time investigating each error report, which equates to large amounts of wasted time. In this way, our model could potentially prove valuable, either on its own, or used in combination with static analysis tools, to reduce the amount of time spent investigating inaccurate error reports.

Despite the high performance of our model on the datasets we examined, there are three limitations to our approach, and we defer addressing these for future work.

- *Detection granularity* In our experiments, we predicted the weakness class that the input program falls into as a graph classification task, i.e., is the entire program vulnerable or not. This limits the practical usefulness of the approach for developers, as generally you need more precise information about where the bug is located. We believe there are at least two possible methods for working around this limitation; the value flow graph for each function (and its callees) can be extracted instead of the entire program, which would achieve function level prediction. Alternatively, we can treat the problem as a node classification rather than graph classification task, but this significantly increases labeling difficulty, which is a challenge within itself.
- *Performance in real-world applications* While we have made some effort to evaluate our model's performance on datasets with a different structure to what it was trained on, the lack of real-world benchmarks for vulnerability detection make it difficult to predict how it would perform if utilized in the practical scenario of analyzing software being written for production use today. We hypothesize that this is an area where more traditional program analysis approaches may perform better, as the semantics and structure of the programs can vary significantly. However, we believe that as datasets for machine learning-based approaches are expanded, this may change.
- *Inconsistent performance across classes* During the Program Representations ablation study (Sect. 8.3), the results of our experiments suggest that different vulnerability types require different representations for both high detection rates, and low false positive rates, as well as being able to differentiate between different, similar vulnerabilities. Future work should attempt to incorporate multiple representations of the program into the graph representation, in place of, or in addition to the instruction embedding we presented in this article.

Funding Open Access funding enabled and organized by CAUL and its Member Institutions. The work has been supported by the Cyber Security Research Centre Limited whose activities are partially funded by the Australian Government's Cooperative Research Centres Programme.

Data availability The datasets generated during and/or analyzed during the current study are available from the corresponding author on reasonable request.

Declarations

Conflict of interest The authors have no financial or proprietary interests in any material discussed in this article.

References

1. The MITRE Corporation: CVE (2022) https://cve.mitre.org/
2. National Institute of Standards and Technology: NVD (2022) https://nvd.nist.gov/
3. K2 Cyber Security Inc (2022) Vulnerabilities up almost 10% in 2021. https://www.k2io.com/the-final-count-vulnerabilities-up-almost-10-in-2021/
4. Rapid7 (2022) Analyzing the attack landscape: rapid7's 2021 vulnerability intelligence report. https://www.rapid7.com/blog/post/2022/03/28/analyzing-the-attack-landscape-rapid7s-annual-vulnerability-intelligence-report/
5. Goseva-Popstojanova K, Perhinschi A (2015) On the capability of static code analysis to detect security vulnerabilities. Inf Softw Technol 68:18–33. https://doi.org/10.1016/j.infsof.2015.08.002
6. Klees G, Ruef A, Cooper B, Wei S, Hicks M (2018) Evaluating fuzz testing. CCS '18, pp. 2123–2138. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3243734.3243804
7. Hanif H, Md Nasir MHN, Ab Razak MF, Firdaus A, Anuar NB (2021) The rise of software vulnerability: taxonomy of software vulnerabilities detection and machine learning approaches. J Netw Comput Appl 179:103009. https://doi.org/10.1016/j.jnca.2021.103009
8. Hovsepyan A, Scandariato R, Joosen W, Walden J (2012) Software vulnerability prediction using text analysis techniques. In: Proceedings of the 4th international workshop on security measurements and metrics, pp. 7–10. https://doi.org/10.1145/2372225.2372230
9. Li Z, Zou D, Xu S, Ou X, Jin H, Wang S, Deng Z, Zhong, Y (2018) Vuldeepecker: a deep learning-based system for vulnerability detection. https://doi.org/10.14722/ndss.2018.23158
10. Harer JA, Kim LY, Russell RL, Ozdemir O, Kosta LR, Rangamani A, Hamilton LH, Centeno GI, Key JR, Ellingwood PM, Antelman E, Mackay A, McConley MW, Opper JM, Chin P, Lazovich T (2018) Automated software vulnerability detection with machine learning. https://doi.org/10.48550/ARXIV.1803.04497
11. Xu A, Dai T, Chen H, Ming Z, Li W (2018) Vulnerability detection for source code using contextual lstm. In: 2018 5th international conference on systems and informatics (ICSAI), pp. 1225–1230. https://doi.org/10.1109/ICSAI.2018.8599360. IEEE
12. Lee Y, Kwon H, Choi SH, Lim SH, Baek SH, Park KW (2019) Instruction2vec: efficient preprocessor of assembly code to detect software weakness with CNN. Appl Sci. https://doi.org/10.3390/app9194086
13. Lattner C, Adve V (2004) Llvm: a compilation framework for lifelong program analysis & transformation. In: International

symposium on code generation and optimization, 2004. CGO 2004., pp. 75–86. https://doi.org/10.1109/CGO.2004.1281665
14. Sanchez-Lengeling B, Reif E, Pearce A, Wiltschko AB (2021) A gentle introduction to graph neural networks. Distill. https://doi.org/10.23915/distill.00033
15. Yang X, Chen Y, Eide E, Regehr J (2011) Finding and understanding bugs in c compilers. In: Proceedings of the 32nd ACM SIGPLAN conference on programming language design and implementation. PLDI '11, pp. 283–294. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1993498.1993532
16. National Institute of Standards and Technology (2017) Juliet C/C++ 1.3 - NIST software assurance reference dataset. https://samate.nist.gov/SARD/test-suites/112
17. MSRC Team (2019) A proactive approach to more secure code. https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/
18. Rapid7 (2018) CVE 100K: by the numbers. https://blog.rapid7.com/2018/04/30/cve-100k-by-the-numbers/
19. Sui Y, Xue J (2016) Svf: interprocedural static value-flow analysis in llvm. In: Proceedings of the 25th international conference on compiler construction, pp. 265–266. https://doi.org/10.1145/2892208.2892235. ACM
20. Sui Y, Ye D, Xue J (2014) Detecting memory leaks statically with full-sparse value-flow analysis. IEEE Trans Software Eng 40(2):107–122. https://doi.org/10.1109/TSE.2014.2302311
21. Cheng X, Wang H, Hua J, Xu G, Sui Y (2021) Deepwukong: statically detecting software vulnerabilities using deep graph neural network. ACM Trans Softw Eng Methodol. https://doi.org/10.1145/3436877
22. Cheng X, Zhang G, Wang H, Sui Y (2022) Path-sensitive code embedding via contrastive learning for software vulnerability detection. In: Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis. ISSTA 2022, pp. 519–531. Association for Computing Machinery, New York, NY, USA . https://doi.org/10.1145/3533767.3534371
23. Andersen LO (1994) Program analysis and specialization for the c programming language. PhD thesis, Citeseer
24. Sui Y, Cheng X, Zhang G, Wang H (2020) Flow2vec: value-flow-based precise code embedding. Proc ACM Program Lang. https://doi.org/10.1145/3428301
25. Alon U, Zilberstein M, Levy O, Yahav E (2019) code2vec: learning distributed representations of code. Proc ACM Prog Lang 3:1–29. https://doi.org/10.1145/3290353
26. llvm-admin team (2022) The LLVM Compiler Infrastructure Project. https://llvm.org/
27. Clang developers (2022) Clang Static Analyzer. https://clang-analyzer.llvm.org/
28. Scarselli F, Gori M, Tsoi AC, Hagenbuchner M, Monfardini G (2009) The graph neural network model. IEEE Trans Neural Netw 20(1):61–80. https://doi.org/10.1109/TNN.2008.2005605
29. CSIRO's Data61 (2018) StellarGraph machine learning library. GitHub. https://github.com/stellargraph/stellargraph
30. Li Y, Zemel R, Brockschmidt M, Tarlow D (2016) Gated graph sequence neural networks. In: Proceedings of ICLR'16. https://doi.org/10.48550/ARXIV.1511.05493
31. Zhou Y, Liu S, Siow J, Du X, Liu Y (2019) Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Advances in Neural Information Processing Systems. Vol. 32
32. Yang X, Chen Y, Eric E, Regehr J (2017) Csmith. https://embed.cs.utah.edu/csmith/
33. Ravitch T (2021) A wrapper script to build whole-program LLVM bitcode files. GitHub. https://github.com/travitch/whole-program-llvm
34. VenkataKeerthy S, Aggarwal R, Jain S, Desarkar MS, Upadrasta R, Srikant YN (2020) Ir2vec: Llvm ir based scalable program embeddings. ACM Trans Archit Code Optim. https://doi.org/10.1145/3418463
35. Bordes A, Usunier N, Garcia-Duran A, Weston J, Yakhnenko O (2013) Translating embeddings for modeling multi-relational data. 26
36. Mikolov T, Chen K, Corrado G, Dean J (2013) Efficient estimation of word representations in vector space. https://doi.org/10.48550/ARXIV.1301.3781
37. Zhang M, Cui Z, Neumann M, Chen Y (2018) An end-to-end deep learning architecture for graph classification. In: Proceedings of the AAAI conference on artificial intelligence. Vol 32
38. Monti F, Frasca F, Eynard D, Mannion D, Bronstein MM (2019) Fake news detection on social media using geometric deep learning. https://doi.org/10.48550/ARXIV.1902.06673
39. Cho K, van Merrienboer B, Gulcehre C, Bahdanau D, Bougares F, Schwenk H, Bengio Y (2014) Learning phrase representations using RNN encoder-decoder for statistical machine translation
40. Fey M, Lenssen JE (2019) Fast graph representation learning with pytorch geometric. arXiv preprint arXiv:1903.02428. https://doi.org/10.48550/ARXIV.1903.02428
41. Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L, Desmaison A, Köpf A, Yang E, DeVito Z, Raison M, Tejani A, Chilamkurthy S, Steiner B, Fang L, Bai J, Chintala S (2019) Pytorch: An imperative style, high-performance deep learning library . https://doi.org/10.5555/3454287.3455008
42. Marjamäki D (2022) Cppcheck - A tool for static C/C++ code analysis. GitHub. https://github.com/danmar/cppcheck/
43. Clang-developers (2022) Clang: a C language family frontend for LLVM. Clang developers. https://clang.llvm.org/index.html
44. Pinconschi E (2020) GitHub - epicosy/devign: effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. https://github.com/epicosy/devign
45. Li Z, Zou D, Xu S, Jin H, Zhu Y, Chen Z (2022) Sysevr: a framework for using deep learning to detect software vulnerabilities. IEEE Trans Depend Secure Comput 19(4):2244–2258. https://doi.org/10.1109/TDSC.2021.3051525
46. Zhen Li DZ, Xu S, Jin H, Zhu Y, Chen Z (2021) GitHub - SySeVR/sysevr. https://github.com/SySeVR/SySeVR
47. Yang S, Yu X, Zhou Y (2020) LSTM and GRU neural network performance comparison study: Taking yelp review dataset as an example. In: 2020 International workshop on electronic communication and artificial intelligence (IWECAI), pp. 98–101. https://doi.org/10.1109/IWECAI50956.2020.00027
48. Li Z, Zou D, Tang J, Zhang Z, Sun M, Jin H (2019) A comparative study of deep learning-based vulnerability detection system. IEEE Access 7:103184–103197. https://doi.org/10.1109/ACCESS.2019.2930578
49. Synopsys, Inc. (2022) Coverity Scan - Static Analysis. https://scan.coverity.com/
50. Roy CK (2009) Detection and analysis of near-miss software clones, pp. 447–450. https://doi.org/10.1109/ICSM.2009.5306301
51. Jang J, Agrawal A, Brumley D (2012) Redebug: finding unpatched code clones in entire OS distributions. https://doi.org/10.1109/SP.2012.13
52. Kim S, Woo S, Lee H, Oh H (2017) Vuddy: a scalable approach for vulnerable code clone discovery. In: 2017 IEEE symposium on security and privacy (SP), pp. 595–614. https://doi.org/10.1109/SP.2017.62. IEEE
53. Ferenc R, Bán D, Grósz T, Gyimóthy T (2020) Deep learning in static, metric-based bug prediction. Array 6:100021. https://doi.org/10.1016/j.array.2020.100021

54. Russell R, Kim L, Hamilton L, Lazovich T, Harer J, Ozdemir O, Ellingwood P, McConley M (2018) Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE international conference on machine learning and applications (ICMLA), pp. 757–762. https://doi.org/10.1109/ICMLA.2018.00120. IEEE
55. Tanwar A, Sundaresan K, Ashwath P, Ganesan P, Chandrasekaran SK, Ravi S (2020) Predicting vulnerability in large codebases with deep code representation. https://doi.org/10.48550/ARXIV.2004.12783
56. Xu R, Tang Z, Ye G, Wang H, Ke X, Fang D, Wang Z (2022) Detecting code vulnerabilities by learning from large-scale open source repositories. J Inf Secur Appl 69:103293. https://doi.org/10.1016/j.jisa.2022.103293
57. Thapa C, Jang SI, Ahmed ME, Camtepe S, Pieprzyk J, Nepal S (2022) Transformer-based language models for software vulnerability detection. ACSAC '22, pp. 481–496. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3564625.3567985
58. Alon U, Brody S, Levy O, Yahav E (2018) code2seq: Generating sequences from structured representations of code. In: International conference on learning representations. https://doi.org/10.48550/ARXIV.1808.01400
59. Sun X, Liu C, Dong W, Liu T (2023) Improvements to code2vec: generating path vectors using RNN. Comput Secur 132:103322. https://doi.org/10.1016/j.cose.2023.103322