VulANalyzeR: Explainable Binary Vulnerability Detection with Multi-Task Learning and Attentional Graph Convolution

LITAO LI*, STEVEN H. H. DING*, YUAN TIAN*, BENJAMIN C. M. FUNG†, PHILIPPE CHARLAND§, WEIHAN OU*, LEO SONG* and CONGWEI CHEN*, * School of Computing, Queen's University, Canada, § Mission Critical Cyber Security Section, Defence R&D Canada, Canada, and † School of Information Studies, McGill University, Canada

Software vulnerabilities have been posing tremendous reliability threats to the general public as well as critical infrastructures, and there have been many studies aiming to detect and mitigate software defects at the binary level. Most of the standard practices leverage both static and dynamic analysis, which have several drawbacks like heavy manual workload and high complexity. Existing deep learning-based solutions not only suffer to capture the complex relationships among different variables from raw binary code, but also lack the explainability required for humans to verify, evaluate, and patch the detected bugs.

We propose VulANalyzeR, a deep learning-based model, for automated binary vulnerability detection, CWE type classification, and root cause analysis to enhance safety and security. VulANalyzeR features sequential and topological learning through recurrent units and graph convolution to simulate how a program is executed. The attention mechanism is integrated throughout the model, which shows how different instructions and the corresponding states contribute to the final classification. It also classifies the specific vulnerability type through multi-task learning as this not only provides further explanation but also allows faster patching for zero-day vulnerabilities. We show that VulANalyzeR achieves better performance for vulnerability detection over the state-of-the-art baselines. Additionally, a Common Vulnerability Exposure (CVE) dataset is used to evaluate real complex vulnerabilities. We conduct case studies to show that VulANalyzeR is able to accurately identify the instructions and basic blocks that cause the vulnerability even without given any prior knowledge related to the locations during the training phase.

CCS Concepts: • **Security and privacy → Software reverse engineering**; • **Computing methodologies → Neural networks**.

Additional Key Words and Phrases: Binary Vulnerability Detection; Multi-Task Deep Learning; Attentional GCNN; Explainability

1 INTRODUCTION

Vulnerability detection is a constantly studied problem in the security community, due to the potential severe hazards being caused by vulnerabilities. Many cyber attacks are rooted in software vulnerabilities. Unlike malware, these vulnerabilities are difficult to detect since they go beyond a simple inference of the program functionalities and require a deep understanding of the complex relationships among the variables. According to Common Vulnerability Exposures (CVE) statistics,

Authors' address: Litao Li*, Steven H. H. Ding*, Yuan Tian*, Benjamin C. M. Fung†, Philippe Charland§, Weihan Ou*, Leo Song*; Congwei Chen*, * School of Computing, Queen's University, 99 University Ave, Kingston, Canada and § Mission Critical Cyber Security Section, Defence R&D Canada, , Valcartier, Canada and † School of Information Studies, McGill University, 845 Rue Sherbrooke O, Montreal, Canada, *{litao.li,steven.ding,y.tian, weihan.ou,leo.song,congwei.chen}@queensu.ca, ben.fung@mcgill.ca, philippe.charland@drdc-rddc.gc.ca.

XXXX-XXXX/2023/4-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

the number of published vulnerabilities has more than doubled from 6,447 in 2016 to 14,714 in 2017[1]. Modern vulnerabilities also become more difficult to manage and discover, and bug patches cannot catch the speed of growing vulnerabilities [1, 2]. Nowadays, software engineering technologies grow more abstractive, and uncontrolled code clone becomes a prevalent issue. The abstraction can hide “unnecessary" details that the programmers overlook and that are leveraged by hackers, thus causing vulnerabilities hard to avoid [3]. Existing dynamic and static approaches for detecting vulnerabilities in both source and binary code have proven to be quite effective, but still have many drawbacks. For example, static-based tools, such as Flawfinder [4] and RATS [5], involve heavy manual work of domain experts, who are constantly in critical shortage in the cybersecurity job market. Dynamic program analysis is also very costly, due to the path explosion problem and could lead to poor program coverage. In comparison, data-driven approaches can make up for the aforementioned disadvantages. Data-driven models usually require far less manual input and feature crafting, since they can learn generalizable patterns from the data with a careful design. These models may take longer in the training stage compared to certain static or dynamic analysis techniques, but can provide far superior speed during inference.

Vulnerability detection is usually conducted at two levels, namely source and binary code levels. Research on source code level has shown promising results [6, 7]. However, since source code is often not available in many scenarios, end-users or edge devices have difficulties and limited resources to determine, evaluate, and verify if a particular program or a 3rd-party library contains vulnerabilities. Binary vulnerability detection then becomes applicable in a wider range of situations. When compared to its source code equivalent, binary code detection is significantly more challenging because of the loss of much syntactical and semantic information during the compilation process from source code.

A few studies have already applied deep learning for vulnerability detection at binary level [8, 9], and the results are quite effective. However, these existing methods can still be hard to use in practice. First, these models simply output a binary classification of either vulnerable or non-vulnerable, without providing any information about the location of the vulnerability. Engineers would still need to track the program variables and path to find the exact location, thus leading to high costs. Another drawback is that the current approaches do not differentiate the types of vulnerability, such as Common Weakness Enumeration (CWE)[2] types during the inference stage. Engineers need to investigate the program based on the types of vulnerabilities and take actions accordingly. If both the location of the detected vulnerability and its CWE type are available as explanations, along with the vulnerability detection output returned by deep learning models, there would be much less human effort involved to validate the model output and derive actionable insights.

We consider the drawbacks of the existing techniques and propose VulANalyzeR, a deep learning-based model for explainable vulnerability detection and CWE classification. To briefly show its importance, consider a sample output of VulANalyzeR on a vulnerable program related to CWE 121-stack-based buffer overflow, as shown in Fig. 1. The model not only predicts the vulnerability and CWE type but also highlights the instructions we should pay attention to in the assembly code. Blue and orange highlighting corresponds to basic blocks and instructions, respectively, and the opacity indicates the score of each element. The scores are calculated from the attention weights, which will be discussed in detail in Section 3.4. The weights are obtained from the neural network model, which is optimized to detect vulnerabilities. The scores can then be interpreted as how much each basic block or instruction contributes to the final classification. To understand the code more

[1]Statistics on Common Vulnerabilities and Exposures (CVE) Details
[2]Common Weakness Enumeration (CWE)

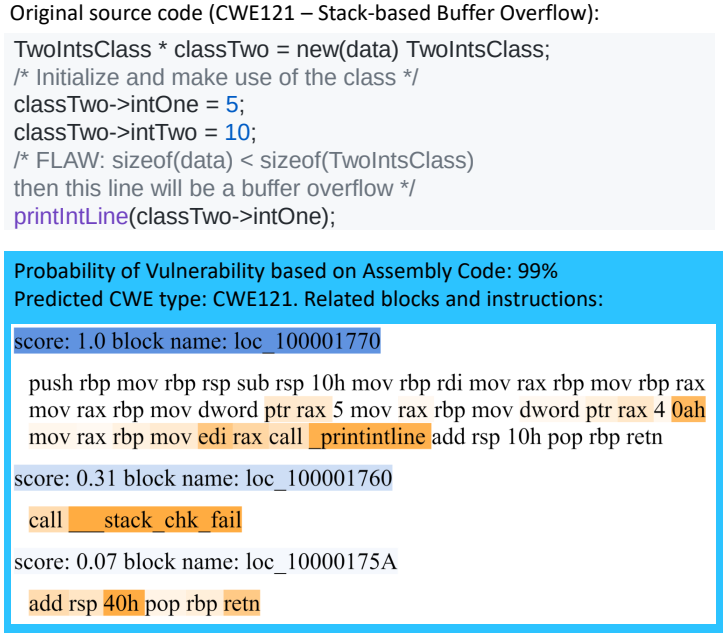

Fig. 1. VulANalyzeR's output highlighting the vulnerable basic blocks and instructions. The vulnerability happens when the data array for one integer has been assigned with two integers, thus causing a buffer overflow (see the source code above). The blue color highlights related basic blocks and the orange color highlights related instructions. The color intensity indicates the score of a basic block or an instruction. In this case, the model highlights the correct block where two integers are assigned to the data object. It also captures `0ah` and `rax` which correspond to the two integers. The size of the stack frame that contains the data array, *40h*, is also highlighted. **VulANalyzeR does not require the source code part and is not given any location related hints in training.**

easily, the source code is shown on top of the figure. A potential buffer overflow could occur when the `sizeof(data)` is less than `sizeof(TwoIntsClass)`. In this case, the array size of the `data` is 1 and `TwoIntsClass` object contains two integers. On the bottom, the block `loc_100001770` has the highest score of 1.0, which indicates that most attention has been paid to this block. This block tries to write two integers into the data buffer, which is exactly where the vulnerability is located. VulANalyzeR also provides a finer granularity - the individual token level. It highlights `0ah`, which is `10` in the source code, as well as the register holding this value. `rax` is highlighted, as it contains 5, the first integer.

To the best of our knowledge, combining both explainability and CWE classification into vulnerability detection has not yet been well studied in the literature at either the binary or source code level. There are previous works that aim at providing explainable results for vulnerability detection [10, 11]. However, these require further perturbed processing and manual selections to extract important tokens or segments from the code, which is inefficient and require additional work besides the model training. VulANalyzeR can combine the training and explainabililty in one architecture and provides an end-to-end learning paradigm. To summarise, we make the contributions as the following:

- Automated and data-driven end-to-end binary vulnerability detection that requires minimal domain knowledge and abandons manual feature crafting.

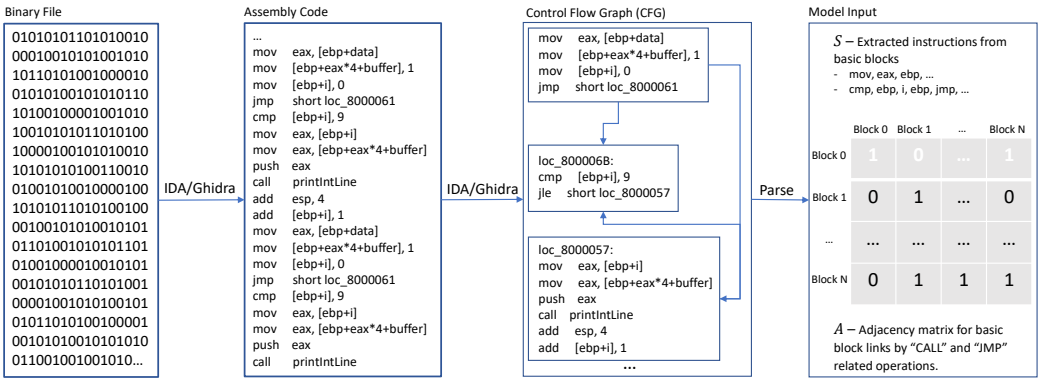

Fig. 2. Binary file processing and transformation for model input. The 3-step data pre-process: disassembling binary files to assembly instructions; parsing and merging CFGs and Call Graphs; and extracting instruction and adjacency matrices.

- Combining sequential learning and topological learning to capture the semantics of instructions and complex structural relationships of binary code, and enhance the detection capability.
- Integrated multi-head attention mechanism to provide interpretable results for security engineers.
- Multi-task learning for classifying CWE types while augmenting vulnerability detection accuracy.
- Benchmarking of state-of-the-art data-driven methods for vulnerability detection and CWE classification on both synthetic datasets and real Common Vulnerability Exposures (CVE).

The rest of the paper is organized as follows. Section 2 defines the problem, including the input, output, and model. We also introduce relevant notations. In Section 3, we elaborate on the technical details of our model design and methodology we used. Section 4 presents our benchmark and case study: dataset description and statistics; experiment setup; results and comparison with the state-of-the-art methods. Section 5 discusses the existing work in dynamic analysis and deep learning methods for binary vulnerability detection. Section 6 concludes this paper. Our code is available at [3].

2 PROBLEM DEFINITION

Before diving into the methodological details, we first formally introduce the related notations and define the studied problem. The input to the overall model is a binary file. $\mathcal{G} = (\boldsymbol{S}, \boldsymbol{A})$ combines all the Control Flow Graphs (CFGs) in the binary file where $\boldsymbol{s} \in \boldsymbol{S}$ denotes the sequence of instructions within a basic block, and $\boldsymbol{A}$ denotes the adjacency matrix where the rows/columns represent nodes, and values represent the existence of an edge. An edge is created between two basic blocks if they are connected in a CFG or in the function call graph. We use a neural network as a function f parameterized by θ to model the relationship between a graph $\mathcal{G}$ and the output signals y. Specifically, there are two output signals: the probability of being vulnerable y_{vul} and the probability distribution over CWE categories y_{cwe}. Overall, given a graph $\mathcal{G} = (\boldsymbol{S}, \boldsymbol{A})$ constructed from a binary file, we apply a learning function $f : \mathcal{G} \rightarrow (\mathbb{R}, \mathbb{R}^{|\text{CWE}|})$ and obtain $y_{\text{vul}}, y_{\text{cwe}}$.

[3] Github Repository

3 METHODOLOGY

In this section, we discuss the 6-component design of VulANalyzeR. For each component, we first discuss the **design concern**, which is the rationale for choosing the design in certain ways. Then, we show the technical details, such as neural network architectures.

The first component f_p pre-processes and transforms binary programs into the input form in which deep learning algorithms can consume and learn from. This can be done by parsing control flow graphs $\mathcal{G}$, to extract instruction information $\boldsymbol{S}$ and graph structures $\boldsymbol{A}$.

$$\boldsymbol{S}, \boldsymbol{A} = f_p(\mathcal{G})$$

The second component f_s models the instructions $\boldsymbol{S}$ of basic blocks, which are sequences of operations and operands. The output is a learned instruction representation $\boldsymbol{V}_s$.

$$\boldsymbol{V}_s = f_s(\boldsymbol{S})$$

The third component f_g aggregates the instruction representation $\boldsymbol{V}_s$ with graph structure information $\boldsymbol{A}$ to learn the graph representation $\boldsymbol{V}_r$ through a graph neural network.

$$\boldsymbol{V}_r = f_g(\boldsymbol{V}_s, \boldsymbol{A})$$

The fourth component f_α implements a specific aggregation mechanism, a multi-head attention that is used in order to explain the model result and locate the vulnerability. The output $\boldsymbol{G}_r$ is a learned graph representation, while the $score$ indicates the amount of attention paid to each instruction token or basic block.

$$\boldsymbol{G}_r, score = f_\alpha(\boldsymbol{V}_r)$$

The fifth component f_c is a multi-task design in order to classify the both the vulnerability y_{vul} and CWE type y_{cwe}.

$$y_{\text{vul}}, y_{\text{cwe}} = f_c(\boldsymbol{G}_r)$$

The last component f_v creates visualization of explainable results from the model output y_{vul}, y_{cwe} and output $score$ from intermediate component.

$$visualization = f_v(y_{\text{vul}}, y_{\text{cwe}}, score)$$

3.1 Preparing Model Input

In order to properly feed data into a neural network, choosing the kinds of data and input format is critical, as it directly influences the performance and learning quality. We have two aspects of design concerns that guide us through on how to transform a binary program into appropriate numerical values.

Design concern 1: Which form of binary program should be used? Deep learning and neural networks are computer programs that take numerical vectors as input. Binary code in its original form consists of numerical values of 0's and 1's, which in theory can be absorbed by algorithms and learned. This form of input is also used in applications such as malware detection [12]. However, it can be difficult for the neural network to learn the functionality and vulnerability patterns, due to the lack of syntactical structure This is especially true for vulnerability detection, where the model needs to understand nuanced relationships among program variables. Also, an analyst would not be able to understand the output, even if the model makes the correct prediction. Therefore, we need to represent the binary code in another form that would enable analysts to understand the semantics and structure of binary code. We choose assembly code as the form to represent binary code. Assembly code can be obtained from object files via dissembling techniques, and it captures the relative syntactical and structural information of binary code.

Design concern 2: How to represent the structure of binary code? Assembly code, as a linear layout of instructions, misses important structures that embed the actual flow of code execution. When modeling source code, existing studies have proposed multiple ways to model source code. One may transform the source code into abstract syntax trees and perform tree-based neural network for classification [13]. Others treat programs as natural language text and then apply off-the-shelf natural language processing techniques to model code in a linear fashion [14]. These representations have been shown to be effective in vulnerability discovery on source code. Recent research that involves assembly code also heavily leverages the graph structure of the assembly code to simulate program execution [15]. In general, the binary file can be broken down into a few levels: A binary file contains multiple assembly functions; each assembly function can be represented as a CFG with interconnected basic blocks; and each block contains non-branching and non-referred instructions. In rule-based vulnerability static analysis, CFGs are analyzed to identify vulnerabilities [16] as well. There are several forms we can choose to represent a binary program: CFGs, data flow graphs, and instructions. Representing the entire file as a list of instructions does not capture the data flow and call relationships in the program. Data flow graphs possess the call relationships, but lack the information of higher level functionality of each element. CFGs are the most suitable in this case, since we can model the program path, while clearly separating the functionality of each basic block.

With the **design concern 1 and 2** in mind, we choose to first use a disassembler to obtain assembly code from binary code. The assembly code is then further processed and parsed into CFGs as $\mathcal{G}$. While each function contains a CFG, we merge them into a large graph by using the function call information. Note that the binary file contains information, such as comments, strings, and file names that should not be fed into a neural network, as they do not contribute to differentiating vulnerable code. These additional pieces of information are discarded, since they can be easily manipulated and cause false positives. Even though we have not used strings in the data segment, which may help identify string formatting vulnerabilities (CWE 134), we still achieve good performance in this category. All basic blocks are extracted from $\mathcal{G}$, and we obtain the instructions $\boldsymbol{s} \in \boldsymbol{S}$ within each basic block. We only include operations and operands from the instruction $\boldsymbol{s}$. To transform the operations and operands into numerical values, we map them into arbitrary index values, and denote the transformed indices as $token_t$ at time step t. The learning of the sequential dependencies between these tokens is further elaborated in Section 3.2. For structural information, we obtain the adjacency matrix $\boldsymbol{A}$ by parsing $\mathcal{G}$ and extracting the "call" statements between the basic blocks. This is used in Section 3.3, where we model the graph structure using a graph neural network f_g. $\boldsymbol{A}$ is important to create the edges in the graph so the structure can be represented. The process for preparing the data input is shown in Fig. 2.

3.2 Modelling Assembly Instructions

After obtaining $\boldsymbol{S}$ and $\boldsymbol{A}$, we first aim at teaching the model to understand the semantics of assembly language and model the sequential dependencies of instructions and variables contained in $\boldsymbol{S}$. This component simulates how humans read and understand assembly code.

Design concern 3: How to learn assembly language semantics and capture dependencies between variables? Assembly code, as a low-level programming language, behaves similarly to natural languages where it is interpreted in a particular sequence. Each token in the assembly code has a particular meaning, and the tokens are not independent. For example, "add" is semantically similar to "addc". To capture their relationships, we need to learn a numeric vector to represent each token's semantic meaning, since treating them as a discrete signal will ignore such relationships. Additionally, similar to natural languages, it is interpreted in a particular context and in a dynamic order. This means the existence of a token impacts the functionality of others in prior or later time,

and the same tokens do not represent the same semantics in different locations, correspond to the program states.

Generally, in deep learning, the relationship among different signals (which is a token in assembly code in our case) is captured by an embedding layer. Each unique discrete token is mapped to a numeric vector to be learned. As mentioned in 3.1, the operations and operands are extracted for each basic block and are in the form of pairs. We concatenate the instructions to obtain a sequence of tokens in the form of [operation_1, operand_1, operation_2, operand_2, ...], denoted as $\boldsymbol{s}$. Different $\boldsymbol{s}$ are expected to have varied lengths. The number of tokens within a basic block is $\boldsymbol{s_i}$: $\boldsymbol{s_i} \in \mathcal{R}^{0,...,|s_i|}$, where $|s_i|$ denotes the length of instruction $\boldsymbol{s_i}$ in the form:

$$\boldsymbol{s_i} = token_1, token_2, token_3, ...token_{|s_i|}$$

An embedding layer is denoted as a function f_e, which is parameterized by a vector for each token jointly trained with the other components of our model. The output of f_e is then the initial embedding vector $\boldsymbol{v}_t$ at time t: $\boldsymbol{v}_t = f_e(t)$. The embedding matrix is also denoted as $\boldsymbol{V}$.

To capture the semantics and the contexts of the instructions as described in **design concern 3**, sequence models are a typical choice to model temporal relationships. We adopt the Recurrent Neural Network (RNN) for this requirement. RNN is designed for sequence learning, where they are able to learn dynamic behavior of sequences in conjunction with the embedding layer. RNN models, such as Long-Short-Term-Memory (LSTM) [17] and Gated Recurrent Unit (GRU) [18] are both the state-of-the-art sequence models, that are able to capture the dependency between input in long sequences, which is a normal characteristic of assembly language. Bi-directional design is beneficial, since it combines two levels of RNNs, one going forward from the beginning and the other going backward from the end. It can mitigate the problem where learning tends to focus more heavily towards the end of the sequence.

We use two bi-directional GRU layers, since in our experiment, GRU outperforms LSTM for both accuracy and training time. GRU [19] updates the state r_t at every time step t by a linear interpolation between previous state r_{t-1} (1) and the candidate activation $\tilde{r}_t$ (3).

$$r_t = (1 - z_t)r_{t-1} + z_t\tilde{r}_t \tag{1}$$
$$z_t = \sigma(\boldsymbol{W}_z v_t + \boldsymbol{U}_z h_{t-1}) \tag{2}$$
$$\tilde{r}_t = tanh(\boldsymbol{W} v_t + \boldsymbol{U}(q_t \odot r_{t-1}) \tag{3}$$
$$q_t = \sigma(\boldsymbol{W}_r v_t + \boldsymbol{U}_r r_{t-1}) \tag{4}$$

Note z_t (2) is an update gate; q_t (4) is a reset gate; $\boldsymbol{W}_z$, $\boldsymbol{U}_z$, $\boldsymbol{W}_r$, $\boldsymbol{U}_r$,$\boldsymbol{W}$, and $\boldsymbol{U}$ are all weight matrices; $\odot$ represents element-wise multiplication; and σ denotes a *sigmoid* function.

The first Bi-GRU layer f_{s1} takes the output from embedding layer v_t and returns the forward embedding matrix $\boldsymbol{R}_{f1}$ and backward embedding matrix $\boldsymbol{R}_{b1}$: (5). The two sequences are concatenated into $\boldsymbol{R}_1$ (7)

$$\boldsymbol{R}_{f1}, \boldsymbol{R}_{b1} = f_{s1}(\boldsymbol{V}) \tag{5}$$
$$\boldsymbol{R}_{f2}, \boldsymbol{R}_{b2} = f_{s2}(\boldsymbol{R}1) \tag{6}$$
$$\boldsymbol{R}_1 = \boldsymbol{R}_{f1} \oplus \boldsymbol{R}_{b1} \tag{7}$$
$$\boldsymbol{R}_2 = \boldsymbol{R}_{f2} \oplus \boldsymbol{R}_{b2} \tag{8}$$

where $\oplus$ denotes concatenation of two matrices. $\boldsymbol{R}_1$ is fed into the 2nd Bi-GRU layer f_{s2}. Similarly, forward embedding matrix $\boldsymbol{R}_{f2}$ and backward embedding matrix $\boldsymbol{R}_{b2}$: (6) are obtained. Eventually the output $\boldsymbol{R}_2$ can be concatenated as shown in (8).

Sometimes, GRU may fail to model the long term dependency. Attention mechanism is able to better capture the dependency between distant words or tokens, as the length of the sequence

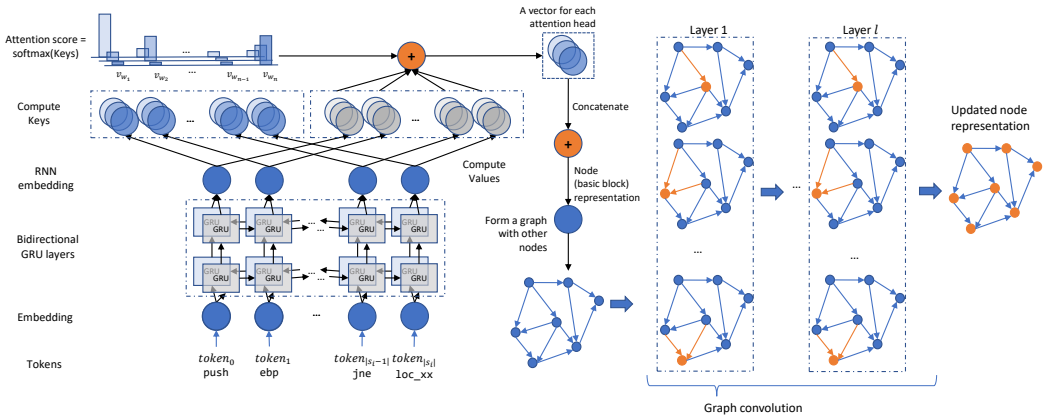

Fig. 3. Neural network design. Tokens are first fed into a embedding layer to produce the embeddings for two stacked bi-directional GRU layers. After, the GRU layer output is aggregated through multi-head attention to produce the node representation. The GCN then consumes the node embedding with adjacency matrix to learn the updated embedding V_r' through multiple convolution layers.

becomes larger by providing a shortcut to connect the each timestamp to the final layer output. More importantly, attention allows analysts to visualize that the same tokens do not carry the same information in different locations, thus providing explainability. The explainability is explicitly discussed in Section 3.4. In our design, we use multi-head self-attention to aggregate the output $\boldsymbol{R}_2$ as shown in Fig. 3.

Multi-head attention is preferred over single-head attention, since multi-head is able to accommodate the required expressiveness of natural languages [20]. The attention layer calculates the key $\boldsymbol{S}_w$ and value $\boldsymbol{S}_v$ separately from $\boldsymbol{R}_2$:

$$\boldsymbol{S}_w = \sigma(\boldsymbol{R}_2 \times \boldsymbol{W}_{s2w}) \tag{9}$$

$$\boldsymbol{S}_v = \sigma(\boldsymbol{R}_2 \times \boldsymbol{W}_{s2v}) \tag{10}$$

Where $\boldsymbol{W}_{s2w}$ and $\boldsymbol{W}_{s2v}$ are weight matrices. The attention score per head for each time step is then calculated through $softmax(\boldsymbol{S}_w) \in [0, 1]$. These attentions scores are extracted directly from this layer to show which instructions contribute most to the results (importance of the instruction tokens). The representation for each node $\boldsymbol{V}_s$ (11) is the dot product of key and value.

$$\boldsymbol{V}_s = \boldsymbol{S}_w \cdot \boldsymbol{S}_v \tag{11}$$

The node representation vector $\boldsymbol{v}_s \in \boldsymbol{V}_s$ is then fed into the graph neural network to learn the structure and interaction among the nodes, which is discussed next.

3.3 Modelling Graph Structure Information

In the last section, we have obtained the representation $\boldsymbol{V}_s$ for basic blocks through the sequence model f_s. However, the control flow graph, which connects these basic blocks and implies the flow of execution, is not yet considered.

Design concern 4: How to model the program execution flow? Sequence models, such as RNN, can potentially capture the full program semantics if the program is always running in a top-down linear order. However, a program's execution flow is more dynamic than linear, and basic blocks in CFGs are linked based on jump-related and call-related operations. Their relationships can be cyclic, and a sequence model cannot consider this aspect. In order to match the structural

nature of code, Graph Neural Network (GNN) [21] should generally provide a better framework in the CFG level, when cyclic control flows happen very often. In particular, graph convolution network (GCN) [22] is a type of GNN that can pass the learned information for one node to its linked neighbor in a recursive manner, simulating the process of passing one program state in a basic block to its linked basic block. Asm2Vec [15] uses random walk to simulate program execution. Instead, we use graph convolution to simulate the dynamics of path traversal which is guided by the content rather than doing this randomly. The information contained in a graph includes both node representation and edge direction. Such information allows the network to simulate execution path and model the functionality at the same time. Thus GNN is more powerful when the input is CFGs.

The main input needed for a GNN is a node feature matrix and adjacency matrix. We use the node representation matrix $\boldsymbol{V}_s$ from GRU layers as a node feature input and the adjacency matrix $\boldsymbol{A}$ is extracted from Section 3.1. It should be noted that in our case, a node is a basic block, which contains non-branching and non-referred instructions. One layer in this case represents one message passing between neighboring nodes for aggregating information and calculates the new graph states. Because of this design, the number of layers represents how many degrees the message is able to propagate to other nodes, thus capturing the overall graph structure.

We use GCN shown on the right side of Fig. 3 to aggregate neighboring node information with multiple layers. In each layer, the information of a node i is propagated by its neighbor through a forward update. The forward update of GCN can be formulated as:

$$h_i^{l+1} = \sigma(\sum_{j \in \mathbb{N}_i^r} g(h_i^l, h_j^l)) \tag{12}$$

where $h_i^l \in \mathcal{R}^{d^l}$ is the hidden state of a basic block i during layer l, with d^l being the hidden dimension of layer l. $\mathbb{N}_i^r$ denotes the set of neighboring nodes of node i which can be obtained from $\boldsymbol{A}_i$. g is a message passing function from block j to block i, and in our case, is a linear transformation. σ is an activation function, in which we used $ReLU$. h^0 is the output vector $\boldsymbol{v}_s$ from the sequence model.

Note that the number of layers indicates the degree that information is propagated in a graph. In smaller graphs, this should intuitively be set to a lower number, since duplicated information is propagated more than once, and hence this could lead to overfitting. We denote the output at the final layer as $\boldsymbol{V}_r \in \mathcal{R}^{n \times d_f}$, which is the node representation from the graph model, while d_f denotes the output dimension in the final layer.

3.4 Explainability and Actionability

Design concern 5: How to define explainability in the context of binary code vulnerability detection? Explainability in deep learning has been a challenging problem, since humans and algorithms do not communicate using the same language. In practical scenarios, explainability is application- and context-dependent. For different applications, one has different ways to present what to explain. For example, in object recognition, one can define explainability of the prediction as the part of the image upon which the model is making the decision. In our context, when a user is trying to validate a vulnerability identified by a machine learning model, we believe two important pieces of information should be available. The first one is the locations of the vulnerability root cause, corresponding to the program states of those particular locations. The second one is the type of vulnerability, corresponding to the inner working of why certain locations are vulnerable. These two pieces of output information provide actionable insights for end-users to evaluate, verify,

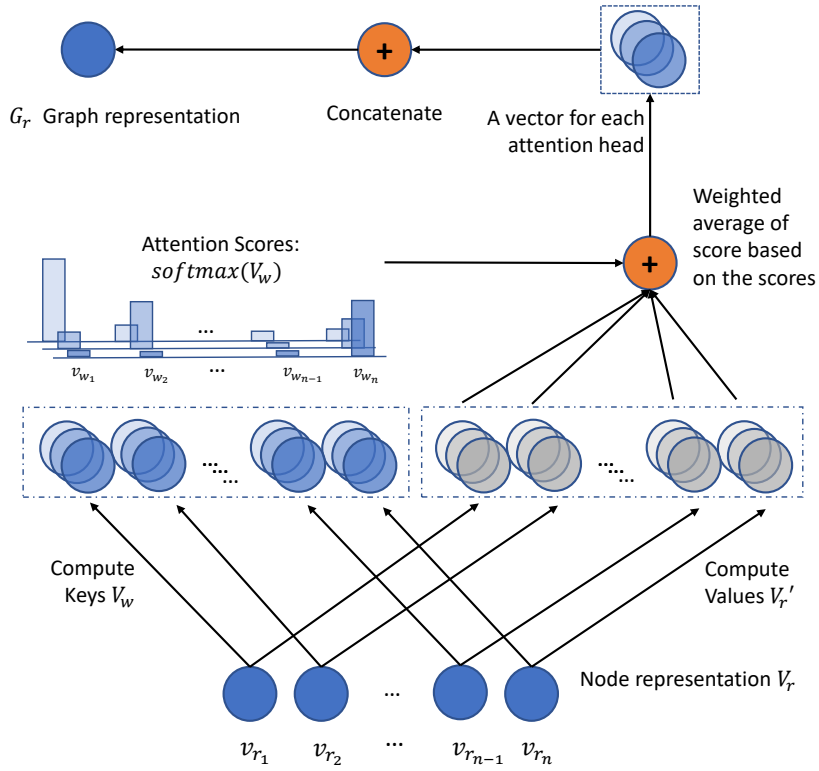

Fig. 4. Weighted Average Multi-Head Attention. G_r can be obtained by the aggregation of node representation V_r through multi-head attention mechanism. Firstly, this component of the model consumes V_r with n nodes,to computes the keys V_w and values V_r'. $softmax(V_w)$ computes the score for each node and the graph representation G_r can be calculated by a weighted average.

and patch the found vulnerability. The locations can be found and highlighted by the attention-based node aggregation method explained later in this section, where the output for the type of vulnerability is achieved by the multi-task learning method presented in Section 3.5.

Design concern 6: Which granularity level should be used to indicate the location of vulnerability? When identifying a vulnerability, one can narrow the focus to a particular token, an instruction, a basic block, or a function. Different granularity levels show different details to analysts and therefore, highlighting the tokens of assembly code on top of the highlighted basic blocks will be the most beneficial, as it gives the most specific information to act upon (see Fig. 1 for an example). It should be noted that the highlighted tokens correspond to the cumulative program state learned through the neural network at that particular location. One highlighted token does not mean that the same token should be highlighted at a different location. For the tokens, we already have a multi-head attention mechanism on the top of the RNN outputs for instruction-level aggregation in Section 3.2. The score matrix $softmax(S_w)$ already indicates which instruction is more important compared to the rest in a basic block. For the basic blocks, which correspond to the nodes in the graph, we also need a mechanism to score and aggregate them into a unified graph-level representation.

With the stated **design concern 6** in mind during the development, we adopt a multi-head attention mechanism with key and value to aggregate the node representations to a graph representation. As shown in Fig. 4, first the value matrix $\boldsymbol{V}_r' \in \mathcal{R}^{(n \times d_G)}$ is calculated through a multi-layer perceptron (MLP) transformation:

$$\boldsymbol{V}_r' = \sigma(\boldsymbol{V}_r \times \boldsymbol{W}_{n2g}) \tag{13}$$

where $\boldsymbol{W}_{n2g} \in \mathcal{R}^{d_f \times d_G}$ is the transformation weight matrix to obtain the node representation with the desired graph embedding dimension d_G. The next step is to obtain the attention weight matrix $\boldsymbol{V}_w \in \mathcal{R}^{d_f \times h}$ for each node with another MLP transformation:

$$\boldsymbol{V}_w = \sigma(\boldsymbol{V}_r \times \boldsymbol{W}_{n2w}) \tag{14}$$

Similar as the last transformation, $\boldsymbol{W}_{n2w} \in \mathcal{R}^{d_f \times h}$ is the matrix to calculate the attention weight for h heads. Then, a softmax for each dimension is applied to obtain a score for each node i. This means:

$$\sum_{i \in nodes} \boldsymbol{V}_{w_i}^{\ j} = 1, \forall j \in h \tag{15}$$

In order to apply the attention weights to the node representation, the shapes need to be consistent. This can be done by forcing the graph representation dimension G_D to be multiples of attention heads h. Then V_r' is reshaped into $V_r' \in \mathcal{R}^{n \times h \times (d_G/h)}$ and computes the weighted average by multiplying V_w along the h dimension. This yields the weighted node representation $V_{weighted} \in \mathcal{R}^{n \times d_G}$. Finally a concatenation or summation can be used to reduce to a graph representation vector $G_r \in \mathcal{R}^{G_D}$. Note that during inference, the attention scores for the basic blocks can be obtained directly from the model weights $softmax(\boldsymbol{V}_w)$.

3.5 Multi-task learning

Our model needs to perform two classification tasks given a particular $\mathcal{G}$, namely vulnerability detection and CWE type classification. This section discusses how we design the model to learn both tasks efficiently and augment both task performance.

Design concern 7: How to work with two different objectives? There are several options to learn multi-task in deep learning. The first method is to learn both tasks independently, which means the knowledge is not shared across the models. The model weights are optimized solely based on one objective. The second design is to combine the labels of CWE and vulnerability to produce 2 $\times$ `number of CWEs`, i.e. the labels become cwe121_good and cwe121_bad. Although this setup allows one set of parameters to be optimized for both tasks, it assumes that cwe121_good and cwe121_bad are completely separated/independent. In fact, they do share information, since both relate to a specific type of program functionality. It is also possible to use soft parameter sharing between models [23], but this approach is prone to over-fitting, which is already likely during original model training. Instead, we can utilize multi-task learning model to combine CWE classification (recognize the category of the functionality) and vulnerability detection with hard parameter sharing [24]. This way, the knowledge can be transferred between functionality recognition and vulnerability detection. The advantage is that one set of weight parameters is learned to perform both tasks simultaneously and is less likely to over-fit to one particular task. In the multi-task learning with hard parameter sharing, CWE labels cannot be further splitted into vulnerable and non-vulnerable labels. This ensures unbiased learning for both vulnerability detection task and CWE classification task.

Given the learned graph representation G_r from component f_α and a learning function f_c, i.e., our neural network, two results can be obtained:

$$y_{\text{vul}}, y_{\text{cwe}} = f_c(G_r)$$

Notice that at the end of a training epoch, two objective functions are integrated, namely binary cross entropy loss (L_{vul}) for vulnerability detection, and a multi-class cross entropy loss (L_{cwe}) for cwe classification. Each loss is then separately back-propagated and all weights are updated. This allows the parameters to reflect both objectives and learn from them.

3.6 Output Visualization for Explainable Classification

In this section, we present how we translate the attention weights to scores for visual highlights. Such results allow for analysts to easily understand the attention paid to each basic block or token in a binary file. Example highlights can be found in Fig 1 and the examples in case studies 4.5.

The attention is defined as a score $\in [0, 1]$ that is associated with a specific element (instruction or basic block). Section 3.2 and Section 3.4 have demonstrated how we can obtain such scores $softmax(\boldsymbol{V}_w)$ (for basic blocks) and $softmax(\boldsymbol{S}_w)$ (for tokens) through multi-head attention mechanism. Given a test sample $\mathcal{G}$, we first feed it into the trained neural network. Then, we directly collect the scores from the corresponding intermediate layers. The higher the score, the more a basic block or instruction contributes to the final classification. We use two separate colors to highlight the scores, where blue highlights basic blocks and orange highlights instructions. The opacity of the highlighted color indicates the magnitude of the score (ranging from 0-1) where more opaque color means higher score. Since the attention weights are based on multi-head, the final score is aggregated using `max` instead of `average` from all heads. The opacity of the highlight color is calculated as:

$$opacity_{block} = \frac{softmax(\boldsymbol{V}_w)}{max(softmax(\boldsymbol{V}_w)} \times 255$$

$$opacity_{token} = \frac{softmax(\boldsymbol{S}_w)}{max(softmax(\boldsymbol{S}_w)} \times 255$$

Note that the sum of scores for all basic blocks or all instructions within a basic block is not then equal to 1, due to the `max` operation. For all the examples shown in this paper, we select the top-3 basic blocks and show all their highlighted tokens.

4 DATASET, EXPERIMENT AND RESULTS

This section describes the datasets used to evaluate VulANalyzeR, including the distribution and statistics of certain aspects of the datasets. A complete experiment set-up is shown in order for others to replicate the experiment. We also discuss the types of metrics used for the evaluation. The experiment is divided into four parts to separately evaluate our contributions, including vulnerability detection, CVE evaluation, CWE classification, and explainability. The datasets in the experiment include the NDSS18 dataset, Juliet Test Suite, and a real world CVE dataset.

The NDSS18 source code dataset is derived from the National Institute of Standards and Technology (NIST): NVD[4] and the Software Assurance Reference Dataset (SARD) project[5]. The NDSS18 source code was first posted [6, 7] and then compiled into binary code [8]. It contains a total of 32,281 binary files. The binary functions can be further split into Windows and Linux platforms,

[4]National Institute of Standards and Technology
[5]Software Assurance Reference Dataset

Table 1. Number of nodes and sequence length distribution (NDSS18)

	mean	std	min	25%	50%	75%	max
Node Count	16	30	2	8	11	16	309
Seqequence Length	17	22	1	1	9	24	231

Table 2. Number of nodes and sequence length distribution (Juliet)

Length	mean	std	min	25%	50%	75%	max
Node Count	117	111	3	81	85	96	1184
Sequence Length	16	16	1	6	10	26	245

and two CWEs (119 399). The labels for vulnerabilities are almost balanced within both the Windows and Linux platform. The second dataset is the Juliet Test Suite[6], which is a synthetic dataset containing 118 different CWEs. A total of 83,624 binary files are in the dataset, where the labels are equally split into vulerable and non-vulnerable files. Table 1 and 2 show that the statistics for sequence length and number of blocks distribution are similar in both datasets.

On top of these two datasets, we also include real Common Vulnerability Exposures (CVE) cases collected by Yaniv et. al. [25]. The corresponding CWEs include CWE 119 and CWE 78, which were ranked 1st and 11th in 2019 according to the CWE archive[7]. There is a total of 8 different CVE's in our testing dataset, which are: cve-2014-0160, cve-2014-6271, cve-2015-3456, cve-2014-9295, cve-2014-7169, cve-2011-0444, cve-2014-4877, and cve-2015-6826. In total, there are 3,379 samples, where 60 of those are vulnerable. The ratio of positive samples in the CVE dataset is less than 2%, which is extremely imbalanced.

4.1 Experiment Setup

Variants of VulANalyzeR. We perform an evaluation for four tasks, namely vulnerability detection, CWE classification, model explainability, and real world CVE detection. We have developed several variants of VulANalyzeR in this section to compare the results between different neural network setups. The First variant is VulRN, which is based solely on RNN without using the GCN to learn the CFG structure. The advantage of using RNN only is the shallower complexity and also faster training time. This is mainly served as a baseline to check if the model is able to learn the vulnerability solely based on the natural language aspect of the program. The next variant is VulGCN, which is similar to VulRN. VulGCN uses only GCN to learn the CFG structure information, without learning the dependency between instructions. However, the embedding layer remains in this setup, in order to transform the index of each token into a more meaningful representation. The node feature for GCN input is aggregated by a 1-dimensional pooling layer from the embedding layer. The third variant is VulRGCN, which is very similar to VulANalyzeR, but without any attention mechanism. The design rationale behind this setup is to have a baseline for sanity check, to ensuring that the addition of attention mechanism does not lower the performance. Note that all three variants still learn multi-task objectives to classify both vulnerability and CWE types.

Evaluation. In both vulnerability detection and CWE classification, we follow the same experimental protocol used in [8, 26], and split both NDSS18 and Juliet datasets as follows: 80% for training, 10% for validation, and 10% for testing, stratified by the vulnerability label. In CVE evaluation, we first evaluate by using the model trained on NDSS18 for direct testing. Then we also show the

[6]Juliet Test Suite

[7]Top-25 Common Weakness Enumeration (CWE)

results by uptraining the model on a small portion, which is 20%, from the CVE's. All evaluation is conducted on the testing set. In terms of vulnerability detection, which is binary classification, we define vulnerable as positive and non-vulnerable as negative. Metrics include *accuracy*, *precision*, *recall*, *F*1*score* and area under the receiver operating characteristic curve (*AUC*).

AUC is calculated by plotting the false positive rate $FPR = \frac{FP}{TN+FP}$ against the true positive rate *TPR* (same as *recall*) set at maximum 200 different thresholds. Since CWE classification is a multi-class classification task, *precision*, *recall*, *F*1, and *AUC* work differently than the binary classification task. In this case, each of the above metrics is first calculated for each class, then the weighted average of all metrics is calculated by the support (numbers of instances in the classes)[8].

Table 3. Hyper-parameter Setup

Hyper-parameters	Set of Value	Best Value	Effect
Batch Size	[16, 32, 64, 128]	64	fastest
Epochs	[10, 30, 50, 70]	50	best AUC
Dimension RNN	[32, 64, 100, 128]	100	best AUC
Dimension GCN	[32, 64, 100, 128]	100	best AUC
Dropout GCN	[0, 0.3, 0.5]	0.2	best AUC
Dropout Final	[0, 0.3, 0.5]	0.3	best AURROC

Hyper-parameters. In terms of hyper-parameter tuning, Table 3 shows the hyper-parameters we choose to investigate during the training process and the best one is obtained through the validation set performance. The metric used to determine the best set of hyper-parameter is validation *AUC*. Batch size contributes little to the model performance, but a greater batch size can significantly reduce training time. However, the largest value we can use without running out of memory is 64. The number of epochs is set to be 50, as it yields the best performance, and a larger number will lead to over-fitting. The hidden dimension for RNN layer is set to be 100 for one direction, which means the bi-directional RNN layer will concatenate the forward and backward to dimension of 200. The hidden dimension for GCN layer is also 100, as it yields the best performance. We also use a dropout rate of 0.2 in the GCN layer and a dropout rate of 0.3 in the final dense layer for better generalization, as these provide a better results in validation set AUC. In terms of other setups, all experiments are conducted on a Windows 10 machine with 32gb ram, one Titan V and Ryzen 9 3900x with 12 cores.

4.2 Vulnerability Detection

In this task, we evaluate the performance of VulANalyzeR using both NDSS18 and Juliet dataset separately. The baselines include the existing state-of-the-art approaches on these two sets of data as well as our variants, VulRN, VulGCN, and VulRGCN.

In terms of existing benchmarks for NDSS18, two papers [8, 26] use the binary NDSS18 for vulnerability detection. We include these binary benchmarks only because we are able to employ the same compiler options and conduct fair comparison. Other source code benchmarks should not be used to direct evaluation, as they use different data processing, which is not available at the binary level. In particular, the binary benchmarks introduce several techniques, which are `MD-CWE`, `MD-CKL`, `MD-RKL`, `MDSAE-NR` and `TDNN-NR`. Since both papers separate the results into Windows, Linux and combined platforms, we present our evaluations in the same way for consistent

[8]sk-learn metrics: precision score

comparison. Table 4 shows the results of all considered approaches on the NDSS18 dataset. In terms of overall performance on both platforms, VulANalyzeR achieves the best accuracy of 89.53% and AUC of 96.79%. In terms of improvement, VulAnalyzer can achieve 13.6%-28.9% higher AUC, and 2.3%-18.9% higher accuracy over five considered benchmarks. Most of the benchmarks have high recall ($\backsim$98%) with lower precision. We consider the balance to be more important as high false positive rates can lead to more expensive cost on the user side, and 94% recall is also not considered a low value. Note that our GCNN model achieves the highest precision score of 87.09%, and the overall AUC and accuracy are almost on par with the state-of-the-art benchmarks. Additionally, Table 5 shows the breakdown of vulnerability detection by the two CWE types for the NDSS18 dataset, where the results are similar and accurate.

Table 4. NDSS18 Vulnerability Detection by Platform

			Windows		
Models	Accuracy	Recall	Precision	F1	AUC
VulANalyzeR	85.19	92.10	80.95	86.17	94.25
VulRN	80.72	77.99	82.56	80.21	91.17
VulGCN	83.12	81.96	**84.07**	83.01	93.64
VulRGCN	85.24	95.09	79.56	86.63	**94.76**
MD-CWE [8]	84.50	97.20	77.70	86.40	84.40
MD-CKL [8]	83.20	97.70	75.80	85.40	83.00
MD-RKL [8]	80.80	86.90	77.60	82.00	80.70
MDSAE-NR [26]	**86.40**	**98.20**	79.50	**89.00**	86.30
TDNN-NR [26]	85.20	97.90	78.30	87.10	85.40
			Linux		
Models	Accuracy	Recall	Precision	F1	AUC
VulANalyzeR	**94.79**	96.65	**93.40**	**95.00**	**98.85**
VulRN	91.03	89.96	92.31	91.18	98.01
VulGCN	90.61	87.79	93.22	90.42	98.01
VulRGCN	93.13	97.50	89.78	93.48	98.83
MD-CWE [8]	86.90	97.80	80.60	88.30	86.80
MD-CKL [8]	85.90	97.20	79.50	87.40	85.70
MD-RKL [8]	82.70	81.30	83.90	82.60	82.70
MDSAE-NR [26]	88.60	**99.10**	84.40	90.20	87.70
TDNN-NR [26]	87.30	98.90	84.10	89.30	87.40
			All		
Models	Accuracy	Recall	Precision	F1	AUC
VulANalyzeR	**89.53**	94.18	85.36	**90.10**	**96.79**
VulRN	85.38	83.47	87.09	85.24	94.89
VulGCN	86.48	84.59	**88.12**	86.32	95.81
VulRGCN	88.79	96.18	83.93	89.64	96.66
MD-CWE [8]	85.30	98.10	78.40	87.10	85.20
MD-CKL [8]	82.30	98.00	74.80	84.00	82.10
MD-RKL [8]	75.30	87.80	70.50	78.20	75.10
MDSAE-NR [26]	87.50	**99.30**	81.20	89.80	87.10
TDNN-NR [26]	86.60	98.70	80.30	88.30	86.30

Table 5. NDSS18 Vulnerability Detection by CWE

			CWE119		
Models	Accuracy	Recall	Precision	F1	AUC
VulANalyzeR	88.96	95.59	85.05	90.01	96.8
VulRN	85.73	84.3	**87.81**	86.02	95.21
VulGCN	**90.25**	95.48	83.25	88.95	**97.87**
VulRGCN	89.33	**96.41**	85.03	**90.36**	97.14
			CWE399		
Models	Accuracy	Recall	Precision	F1	AUC
VulANalyzeR	**88.15**	**95.0**	79.99	**86.8**	**96.16**
VulRN	83.14	76.66	81.17	78.85	92.74
VulGCN	85.89	83.23	**88.87**	85.96	95.59
VulRGCN	85.38	94.35	75.91	84.13	95.84

Table 6. Juliet Vulnerability Detection (CWE121 Only)

Models	Accuracy	Recall	Precision	F1	AUC
VulANalyzeR	99.68	**100.0**	99.38	99.69	99.99
VulRN	96.81	98.44	95.48	96.94	99.03
VulGCN	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**
VulRGCN	99.04	99.68	98.46	99.07	99.96
i2v/CNN [27]	87.6	NA	NA	NA	NA
i2v/TCNN [27]	96.1	NA	NA	NA	NA
w2v/CNN [27]	87.9	NA	NA	NA	NA
w2v/TCNN [27]	94.2	NA	NA	NA	NA
i2v [28]	96.81	97.07	96.65	96.85	NA
bin2img [28]	97.53	97.05	97.91	97.47	NA
w2v [28]	96.01	96.07	95.92	95.99	NA
gcn [29]	97	NA	NA	NA	NA

Table 7. Juliet Vulnerability Detection Result (All CWE)

Models	Accuracy	Recall	Precision	F1	AUC
VulANalyzeR	99.54	**99.85**	99.24	99.54	99.93
VulRN	94.87	95.28	94.49	94.89	98.65
VulGCN	**99.69**	99.59	**99.80**	**99.69**	**99.96**
VulRGCN	98.64	99.32	97.99	98.65	99.78

The Juliet dataset has a different evaluation setup than NDSS18, since the Juliet is much larger, and all of the existing benchmarks only cover a portion of the CWEs within the dataset. We will first show the overall performance on the entire dataset, which totals 83,624 files. Table 7 shows the VulANalyzeR and all variants perform well across all metrics. VulANalyzeR and GCNN both achieved over 99.5% accuracy and over 99.9% AUC. Even though the Juliet dataset is a synthetic

Table 8. Vulnerability Detection - Comparison with VYPER

	CWE121		CWE134		CWE78	
Models	TP rate	TN rate	TP rate	TN rate	TP rate	TN rate
VYPER[30]	0.00	99.00	**100.0**	**100.0**	73.00	**100.0**
VulANalyzeR	**99.38**	**100.0**	**100.0**	99.11	**99.34**	98.75
	CWE122		CWE415		Average	
Models	TP rate	TN rate	TP rate	TN rate	TP rate	TN rate
VYPER[30]	14.00	98.00	**100.0**	**100.0**	57.40	**99.40**
VulANalyzeR	**100.0**	97.95	98.86	**100.0**	**99.52**	99.12

dataset, we still consider the model to be very successful at learning the vulnerability patterns. In the benchmarks from [27, 28], only CWE121 is shown, where the authors use Instruction2Vec (i2v), CNN and Word2Vec (w2v), and have reported 5 versions, namely i2v/CNN, i2v/TCNN, w2v/CNN, w2v/TCNN, and bin2img. The evaluation of CWE121 can be found in Table 6, where VulANalyzeR, VulGCN and VulRGCN all achieved over 99% accuracy and 99% AUC. VulGCN in particular has correctly identified all CWE121 without any false positives or false negatives. Another benchmark is [29], which uses a graph convolution network (gcn) similar to VulGCN, but with different methods for constructing the feature matrix. The author reports the overall accuracy of 97% for 30 different CWEs, where the model is trained on each individual CWE. Note that VulANalyzeR is trained with all CWEs combined and still achieves the overall accuracy of 99.54%.

We also include a comparison with a dynamic analysis-based technique for vulnerability detection. VYPER [30] is a method based on concolic execution of binary code and is used for comparison with VulANalyzeR. In the experiment, VYPER reports 5 different CWEs from the Juliet Test Suite. However, they only include the true positive rate (TPR) and true negative rate (TNR) as metrics. Since we cannot directly calculate the other metrics, TPR and TNR are used for comparison. Table 8 shows the performance on 5 different CWEs. We can observe that although VYPER is able to achieve a high TNR across all CWEs, the reported TPR (or recall) on CWE121 and CWE122 is only 0% and 14% respectively, which indicates its problem in identifying vulnerable programs. VulANalyzeR, by contrast, consistently achieves a very high average TPR of 99.52% and average TNR of 99.12% at the same time for all CWEs.

4.3 Real World CVE Detection

On top of the evaluation on two synthetic datasets, we evaluate the performance of VulANalyzeR on real world CVE dataset. The CVE dataset is significantly more complex and larger compared to the training dataset. To evaluate, we uptrain the previous NDSS18 model using only 20% of CVE samples. We want to test the model's ability to quickly adapt to other types of vulnerabilities. This is a valid approach since in our opinion, many existing CVE's are extremely useful to help identify patterns and used as uptraining resources. We report that the AUC is 83.54%, the accuracy is 99.31%, the recall is 63.89%, the precision is 95.83%, and the F1 score is 76.67%. Considering that the dataset is very imbalanced with less than 2% vulnerable samples, we believe that the model is able to capture existing CVE's. There are very few false positives, which indicates that the model is efficient at identifying vulnerability. Note that even though the accuracy is nearly 100%, it is due to the imbalance of the dataset, and the model has predicted many negatives. There are more false negatives than false positives, which can be a future direction for improvement.

4.4 CWE Classification

This part illustrates the evaluations for multi-class classification of CWE types. Note that the CWE classification shares the same model parameters as vulnerability detection (hard parameter sharing) as we design it to be a multi-task learning model. Since no benchmarks are currently available for CWE classification, we compare VulANalyzeR with our own variants, including VulRNN, VulGCN, and VulRGCN. Table 9 shows the NDSS18 evaluation on CWE classification. Since NDSS18 only contains CWE119 and CWE399, the task is effectively a binary classification task. VulRGCN is able to achieve the best for all 5 metrics, mostly at around 99.9%. We consider all models to be effective at this task with over 99% accuracy and since CWE119 is the most common vulnerability, having the ability to detect CWE119 is valuable in real world scenarios.

Table 9. NDSS18 CWE Type Classification

Models	Accuracy	Recall	Precision	F1	AUC
VulANalyzeR	99.60	99.60	99.60	99.60	99.80
VulRN	99.1	99.1	99.1	99.1	98.42
VulGCN	99.68	99.68	99.68	99.68	99.79
VulRGCN	**99.77**	**99.77**	**99.77**	**99.77**	**99.93**

The Juliet dataset contains 118 different CWE types, which provides a wider coverage of various vulnerabilities. Table 10 shows the results and this time, VulANalyzeR and VulGCN both have all 5 metrics above 90%, and achieve around 33% higher accuracy than VulRN and VulRGCN.

Table 10. Juliet CWE Type Classification

Models	Accuracy	Recall	Precision	F1	AUC
VulANalyzeR	91.61	91.61	94.52	92.52	99.88
VulRN	69.74	69.74	79.63	73.45	98.15
VulGCN	**92.63**	**92.63**	**94.63**	**93.26**	**99.91**
VulRGCN	68.42	68.42	82.42	72.65	98.59

The performances of VulAnalyzer and its variants on two tasks, i.e., vulnerability detection and CWE classification, are reversely related in the Juliet and NDSS18 datasets. In NDSS18, the accuracy is very high (over 99%) for CWE classification, but lower (slightly under 90%) for vulnerability detection. On the other hand, the model accuracy and AUC for the Juliet dataset are both near 100% for the vulnerability detection task, but lower for CWE classification, compared to NDSS18. Since NDSS18 is a more realistic dataset, vulnerability patterns are harder to generalize. The relatively lower accuracy (around 90%) for CWE label classification in Juliet could be due to the imbalance of CWE labels distribution, as many CWEs only have less than 100 samples. In the next part, we further discuss on how to utilize all the outputs from VulANalyzeR in order to investigate vulnerabilities in binary files.

4.5 Case Study: Explainability

We show several case studies to validate the effectiveness of attention mechanism in vulnerability explainability. We compare the highlighted code areas, which are determined by the attention scores, with the real patched code location using the Juliet dataset. We also manually validate

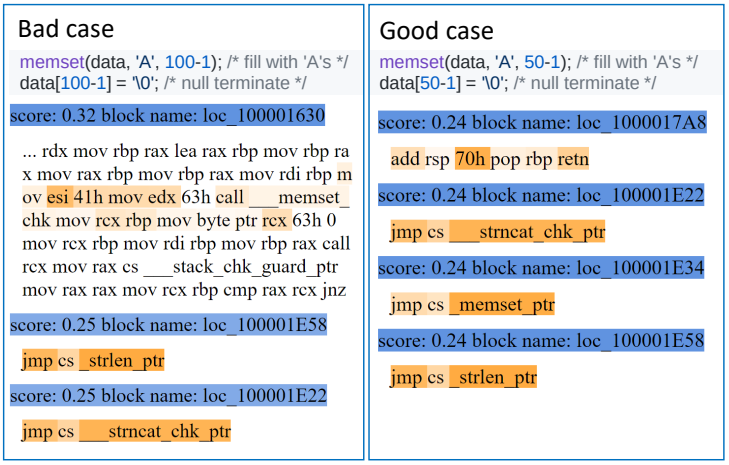

Fig. 5. VulANalyzeR result with highlighted basic blocks and instructions: CWE121 Character Allocation

the model output based on the correctness of the vulnerability detection, CWE classification, and locality.

We randomly pick three target vulnerable files for our case study, each of which contains one of the three major CWE types in the Juliet dataset, i.e., CWE121, CWE416, and CWE78. These three CWE types were chosen, since they are rated as the top ones[9] in 2019, among all CWEs in the Juliet dataset. The Juliet dataset is synthetic and therefore, allows us to understand the vulnerabilities more easily. Similar to our result demonstration in the introduction, we show the corresponding source code for these binary files, since it is readily available and can also be used to validate if the results are indeed helpful. However, it should be noted that the original source code is much longer than what we show in the paper. Due to the space limit, we only show the basic blocks that directly lead to the vulnerabilities. On top of the vulnerable code, we also show the corresponding patched cases for comparison. In each figure, the basic blocks are highlighted in blue, while the instructions are highlighted in orange. The opacity indicates the score of an element, where a more opaque color means a higher score. Recall from Section 3.4 that the score is calculated from $Softmax(V_w)$ and obtained from the intermediate layer. Since these scores are optimized through back-propagation, higher scores then directly contribute more to the final classification.

4.5.1 CWE121: Stack-Based Buffer Overflow. Fig. 5 shows part of a program associated with CWE121 (i.e., stack-based buffer overflow), where the program tries to allocate an array filled with "A"s to a buffer *data* with `len(data) = 50`. The vulnerable case would fill the buffer with 100 "A"s, which is longer than the buffer size, thus causing an overflow. The patched code instead only assigns the array with a length less than or equal to the buffer size. In the model output for the vulnerable code, the block `loc_100001630` has the highest score and this is the block that writes the values into the buffer. The model is able to identify the following highlighted instructions representing both the content and the register that holds the length of the content.

```
mov edx 63h
mov byte ptr rcx 63h
```

[9]CWE Top 25 2019

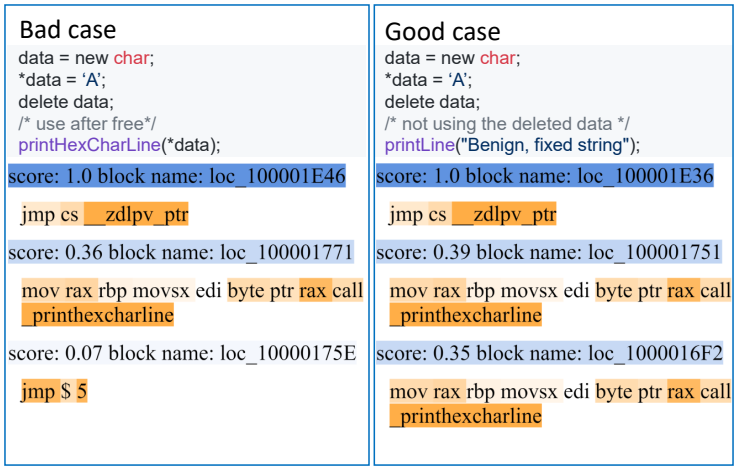

Fig. 6. VulANalyzeR result with highlighted basic blocks and instructions: CWE416 Use-After-Free

The context of instructions are important, as the model does not simply highlight the same instruction everywhere it appears. For example, `mov` appears multiple times within the basic block, but only the ones that impact the vulnerability have higher scores. This implies the model is able to learn the overall state of the program and embed such information into the hidden dimension. The second and third highest block in the bad case are blocks that jump to C++ functions like `strlen` and `strncat`. These two blocks are more related to CWE classification than vulnerability detection, since the model optimizes these two objectives simultaneously. On the right hand side, the model captures `strncat` and `memset` with equal scores in the good case. The model tends to explain the classification of CWE121 while focusing less on explaining why the code is non-vulnerable in this particular case.

4.5.2 CWE416: Use-After-Free. The second case study looks at a program associated with CWE416: use-after-free vulnerability, where a data array is used again after it has been deleted (freed) from the system. Fig. 6 shows both the good and bad cases, with highlighted vulnerability-related code identified by VulANalyzeR. As seen in the bad case, data is used again for printing out after having being deleted from the system, while the good case avoids this. Taking a look at the model output from the bad case, it first highlights the `zdlpv` operation, which corresponds to `delete` in the source code. This already indicates that the vulnerability is related to a deleted variable. The third highest block is simply a `jmp` operation, but if we track down the destination, this block jumps to another block, where the deleted data array is used again. This helps the analyst limit the possible options to only a few CWEs. In fact, this also leads to the root of the vulnerability with some simple analysis.

4.5.3 Case Study 3: CWE78: OS Command Injection. The third example is related to CWE78: OS command injection vulnerability, as shown in Fig. 7. By glancing at the source code for the good and bad cases, the difference is that the command `data` is hard coded in the good case, while it is read from a file in the bad case. One can maliciously inject code into the system through the command. In order to distinguish the difference between the vulnerable and non-vulnerable code in this case, the model needs to first learn about the nature of the vulnerability. At the bottom of the figure, both the bad and good case results assign the highest scores to the blocks where `popen`

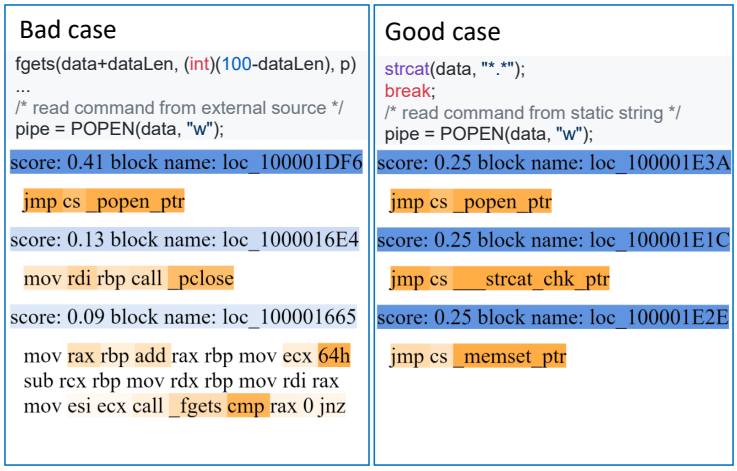

Fig. 7. VulANalyzeR result with highlighted basic blocks and instructions: CWE78 OS Command Injection

is jumped to, indicating that the functionality of the program relates to the systm process. The result for the bad case identifies `fgets` in the third highest block, which hints that the vulnerability is associated with a command read from file. The result for the good case captures `strcat`, which is the program state that copies the fixed string to the buffer. The results in this case are able to clearly identify the exact location of the vulnerability, as well as its type. The explanation is also easily interpretable by an analyst.

These three case studies do not contribute in the same way to help an analyst interpret the results, as the nature of the problem is different among the vulnerabilities. The first case (CWE121) requires an analyst to compare numbers between buffer size and data length, which is harder to capture at first glance, since the results are in assembly code. However, knowing that the CWE type is a buffer overflow helps to narrow down the scope. The second case (CWE416) requires an extra step to track down another basic block required to identify the exact problem. Again, the CWE classification provides such a hint. The CWE78 case is more obvious. The highlighted basic blocks and instructions directly provide explanations to both vulnerability and CWE classifications. Overall, the combination of basic block and instruction level attention provides more intuitive results than considering only one granularity level of attention.

Overall, VulANalyzeR is shown capable of accurately detecting binary vulnerabilities and their root causes. An important question is whether our model can detect more complex vulnerabilities, which are common in real world scenarios. Since synthetic datasets are simple and convenient to obtain, we mostly train the model in this study to demonstrate the capability of our model design. For more complex patterns, VulANalyzeR can reach a desired level of efficacy by uptraining the model using small amount of data. In section 4.3, our model can output an AUC of 83.54% even with a heavy class unbalance in the CVE dataset, which is a complicated and large dataset. Additionally, VulANalyzeR can also locate the trigger conditions of detected vulnerabilities by combining both the root cause and the predicted CWE type. Analysts can track down the associated variables and operations, given the knowledge of what the vulnerability is and where it takes place in the code.

5 RELATED WORK

This section discusses the related work for vulnerability detection in terms of both binary and source code. The survey [31] discusses the vulnerability detection in both source and binary code. It extensively shows two systems, namely Mayhem [32] and Mechanical Phish, which uses Driller [33]. Both methods are categorized as dynamic analysis and suffer from the trade-off between coverage and precision. Moreover, scalability and performance is also hard to achieve using these methods.

We also surveyed research on source code vulnerability detection, since some of the techniques are able to provide design process and ideas for binary code. VulDeePecker [6] is a system that splits source code into “code gadgets", which are few lines of code semantically related to each other. They use the “code gadgets" to feed into a Bi-LSTM network to learn the representation and predict the vulnerable “code gadgets". Although this approach targets source code, it inspired us to use a more granular level for assembly codes. which can be basic blocks instead of functions. Another paper [7] suggests the combination of source code and build process information. The drawback is that the source code is not always compilable, hence the difficulty is using this system. Deep representation was also researched at the binary [8] and source code level [9]. As a drawback, these approaches can lose the ability to explain the results of how vulnerable code is different from non-vulnerable code. Another paper extracts features from CFGs and DFGs, and uses similarity measures to identify vulnerabilities [34]. While this approach tackles the generation of semantics at the binary level, the detection is purely based on cosine similarity, which does not provide explainable results. Another work introduces a tool to embed graph using neural networks [35]. Although the domain and task are different, we can borrow the idea and use a similar approach to embed the CFGs for binary code. Explainable source code vulnerability detection has been studied recently [10, 11]. Both papers are similar and use perturbed samples to identify important parts of the source code that lead to the causes or indicate the patterns of vulnerabilities. The drawback of these methods is that they require additional processing or training on top of the vulnerability detection, which lead to inefficiency and a lack of robustness. Moreover, there is currently a complete lack of explainable binary vulnerability detection methods available.

We further investigated more generalized deep learning-based techniques. Firstly, for modeling instructions, the techniques should apply to natural language-like data for learning instruction information. RNNs, such as LSTM and GRU as well as their variants have been considered [36, 37]. CNN and textCNN are also implemented for vulnerability detection [7, 38]. Additional techniques, such as initial embedding used by the aforementioned papers including word2vec [39] can be helpful. Graph Neural Networks (GNNs) focuses less on vulnerability detection, as these techniques usually only apply to graph structured data. The following specific implementation were investigated to understand how we can utilize GNNs. We conduct research on a survey for GNN [40], which is the use of end-to-end graph learning aiming at performing different kinds of graph operations. GNNs have many variants, but a lot of them have the same underlying structure. Some interesting works include adversarial graph learning [41] and inductive graph representation learning [42]. In particular, previous works utilize GNN combined with other networks for code similarity and vulnerability detection, based on graph structure data, such as CFGs [43, 44].

6 CONCLUSION

Automated binary vulnerability detection is an important task for many security applications, from organization to edge device security. With the advancement of deep learning, data-driven approaches allow for a faster and more scalable detection. However, existing data-driven models still lack good performance and more importantly, the ability to explain the results. We therefore have designed VulANalyzeR, which is a deep learning approach to tackle this problem. On top of

vulnerability detection, which is a binary classification task, we also implemented new features in our model, including explainability, vulnerability localization, and CWE classification. These features are very valuable to analysts in order to better understand the vulnerability and find solutions more efficiently. Specifically, the result explanations provided by VulANalyzeR are in two folds. Firstly, VulANalyzeR identifies and highlights the specific blocks and instructions, that are related to the vulnerability to support vulnerability root cause analysis. Secondly, VulANalyzeR provides CWE type for the detected vulnerability to help engineers verify the model output and take actions accordingly. The above explanations are technically achieved by attention mechanism and multi-task learning. While the added features are successful, as shown in the experiments, we also achieved 2.3%-18.9% higher accuracy and 13.6%-28.9% AUC for vulnerability detection over the state-of-the-art benchmarks for both datasets (NDSS18 and Juliet Test Suite). More importantly, our approach is able to detect real world CVE's, which are significantly more complex and imbalanced than the synthetic benchmarks. Our case studies showing the highlighted locations accurately reflect the root causes of the corresponding vulnerabilities, thus reducing manual efforts in locating and verifying vulnerabilities in the binary code.

The current state of VulANalyzeR also possesses a few limitations. The first limitation is the inability to classify a full range of CWE types, due to the incomplete datasets used during training. Although VulANalyzeR is able to detect the most impactful CWE types, the number of CWE types for NDSS18 and Juliet amounts to 120, which is less than the existing total number. Another limitation is that the scores used for highlighting tokens or basic blocks can contribute to either vulnerability detection or CWE classification task, without the guarantee of being equally distributed. This is due to the nature of back-propagation during neural network training, where the model parameters are adjusted with different values, based on the loss of objectives.

ACKNOWLEDGMENTS

This research is supported by Defence Research and Development Canada (contract no. W7701-176483/001/QCL).

REFERENCES

[1] N. Alexopoulos, S. M. Habib, S. Schulz, and M. Mühlhäuser, "The tip of the iceberg: On the merits of finding security bugs," *ACM Transactions on Privacy and Security (TOPS)*, vol. 24, no. 1, pp. 1–33, 2020.

[2] K. A. Farris, A. Shah, G. Cybenko, R. Ganesan, and S. Jajodia, "Vulcon: A system for vulnerability prioritization, mitigation, and management," *ACM Transactions on Privacy and Security (TOPS)*, vol. 21, no. 4, pp. 1–28, 2018.

[3] D. Gollmann, "Software security–the dangers of abstraction," in *IFIP Summer School on the Future of Identity in the Information Society*. Springer, 2008, pp. 1–12.

[4] "Flawfinder home page," Available at http://https://dwheeler.com/flawfinder/.

[5] "Rough auditing tool for security," Available at http://https://github.com/andrew-d/rough-auditing-tool-for-security.

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

[7] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood *et al.*, "Automated software vulnerability detection with machine learning," *arXiv preprint arXiv:1803.04497*, 2018.

[8] T. Le, T. Nguyen, T. Le, D. Phung, P. Montague, O. De Vel, and L. Qu, "Maximal divergence sequential autoencoder for binary software vulnerability detection," in *International Conference on Learning Representations*, 2018.

[9] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 783–794.

[10] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," *arXiv preprint arXiv:2106.10478*, 2021.

[11] D. Zou, Y. Zhu, S. Xu, Z. Li, H. Jin, and H. Ye, "Interpreting deep learning-based vulnerability detector predictions based on heuristic searching," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 2, pp.

1–31, 2021.
[12] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas, "Malware detection by eating a whole exe," in *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.
[13] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 87–98.
[14] F. Wu, J. Wang, J. Liu, and W. Wang, "Vulnerability detection with deep learning," in *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*. IEEE, 2017, pp. 1298–1302.
[15] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 472–489.
[16] Y. Zheng and X. Zhang, "Path sensitive static analysis of web applications for remote code execution vulnerability detection," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 652–661.
[17] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with lstm," *Neural computation*, vol. 12, no. 10, pp. 2451–2471, 2000.
[18] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.
[19] ——, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.
[20] Z. Yang, Z. Dai, R. Salakhutdinov, and W. W. Cohen, "Breaking the softmax bottleneck: A high-rank rnn language model," *arXiv preprint arXiv:1711.03953*, 2017.
[21] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2008.
[22] R. v. d. Berg, T. N. Kipf, and M. Welling, "Graph convolutional matrix completion," *arXiv preprint arXiv:1706.02263*, 2017.
[23] S. Ruder, "An overview of multi-task learning in deep neural networks," *arXiv preprint arXiv:1706.05098*, 2017.
[24] R. Caruana, "Multitask learning," *Machine learning*, vol. 28, no. 1, pp. 41–75, 1997.
[25] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," *Acm Sigplan Notices*, vol. 51, no. 6, pp. 266–280, 2016.
[26] M. A. Albahar, "A modified maximal divergence sequential auto-encoder and time delay neural network models for vulnerable binary codes detection," *IEEE Access*, vol. 8, pp. 14 999–15 006, 2020.
[27] Y. J. Lee, S.-H. Choi, C. Kim, S.-H. Lim, and K.-W. Park, "Learning binary code with deep learning to detect software weakness," in *KSII The 9th International Conference on Internet (ICONI) 2017 Symposium*, 2017.
[28] Y. Lee, H. Kwon, S.-H. Choi, S.-H. Lim, S. H. Baek, and K.-W. Park, "Instruction2vec: Efficient preprocessor of assembly code to detect software weakness with cnn," *Applied Sciences*, vol. 9, no. 19, p. 4086, 2019.
[29] S. Arakelyan, C. Hauser, E. Kline, and A. Galstyan, "Towards learning representations of binary executable files for security tasks," *arXiv preprint arXiv:2002.03388*, 2020.
[30] E. H. Boudjema, S. Verlan, L. Mokdad, and C. Faure, "Vyper: Vulnerability detection in binary code," *Security and Privacy*, vol. 3, no. 2, p. e100, 2020.
[31] T. N. Brooks, "Survey of automated vulnerability detection and exploit generation techniques in cyber reasoning systems," in *Science and Information Conference*. Springer, 2018, pp. 1083–1102.
[32] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 380–394.
[33] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, no. 2016, 2016, pp. 1–16.
[34] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: a semantic learning based vulnerability seeker for cross-platform binary," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 896–899.
[35] J. Schrouff, K. Wohlfahrt, B. Marnette, and L. Atkinson, "Inferring javascript types using graph neural networks," *arXiv preprint arXiv:1905.06707*, 2019.
[36] F. Wu, J. Wang, J. Liu, and W. Wang, "Vulnerability detection with deep learning," in *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*. IEEE, 2017, pp. 1298–1302.
[37] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *arXiv preprint arXiv:1807.06756*, 2018.
[38] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2018, pp. 757–762.

[39] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.
[40] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *arXiv preprint arXiv:1901.00596*, 2019.
[41] H. Wang, J. Wang, J. Wang, M. Zhao, W. Zhang, F. Zhang, W. Li, X. Xie, and M. Guo, "Learning graph representation with generative adversarial nets," *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 8, pp. 3090–3103, 2019.
[42] R. A. Rossi, R. Zhou, and N. K. Ahmed, "Deep inductive graph representation learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 32, no. 3, pp. 438–452, 2018.
[43] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 363–376.
[44] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems*, 2019, pp. 10 197–10 207.