Received: December 3, 2024. Revised: December 22, 2024.

INASS

International Journal of
Intelligent Engineering & Systems

http://www.inass.org/

A Deep Learning Approach for Binary Code Similarity Detection to Detect Vulnerabilities in Firmware Binaries

Nandish M[1]* **Jalesh Kumar[1]**

[1]*Department of Computer Science and Engineering, JNNCE Shivamogga,*
Visvesvaraya Technological University, Belagavi – 590018, India
* Corresponding author's Email: nandish.m@jnnce.ac.in

Abstract: Connected devices are increasingly widespread and provide substantial advantages, yet they also face significant security challenges, making them susceptible to malware, denial-of-service (DoS), and network attacks, which can result in various issues such as data breaches, system malfunctions, and large-scale disruptions. Vulnerability detection in the connected devices is essential to safeguard against the myriad of security threats posed by the increasing number and diversity of connected devices. The implementation of robust vulnerability detection strategies helps to protect sensitive data, ensure the integrity of critical infrastructure, and maintain consumer trust. In this work, a novel vulnerability detection framework is proposed using graph attention network (GAT) and decoding-enhanced bidirectional encoder representations from transformers with disentangled attention (DeBERTa) model. GAT is used for analyzing complex device networks and detecting vulnerabilities arising from device interactions. DeBERTa's advanced contextual understanding and feature extraction techniques help to detect complex vulnerabilities in connected devices, and its ability to handle diverse data makes it ideal for creating high-quality embeddings. Hence transformer-based language model and GAT are used to generate device binary embeddings. Binary embeddings are used in a generative adversarial networks model to classify it as vulnerable or normal. The dataset considered for the experiment, consists of 18 real-world vulnerabilities across 9 IoT firmware packages. The research findings show that proposed approach achieves good accuracy (by an average of 97.3%) based on real-world vulnerabilities across different firmware packages. Comparative analysis carried out shows the effectiveness of the proposed approach on existing vulnerability detection solutions, IoTSim, DeepWukong and Robin by substantially reducing the frequency of false alarms.

Keywords: Graph attention network, Neural network, Internet of things, Security, DeBERTa.

1. Introduction

The fast growth in interconnected IoT technologies and devices, has also exposed significant security vulnerabilities, necessitating advanced approaches for robust vulnerability detection. Weak devices are used to breach connected networks, giving hackers access to user credentials and confidential company information, thereby affecting IoT device security. IoT devices typically lack the internal security measures needed to thwart security threats. Common flaws [1] and vulnerabilities [2] allow cybercriminals to gain access to a device and use it as a platform to launch sophisticated cyberattacks. Firmware is essential for the operation, communication, security, and efficiency of IoT devices. It bridges the gap between the hardware capabilities of the device and the higher-level applications that use the device's data and functionalities. The continuous development and updating of firmware are critical to maintaining the performance, security, and interoperability of IoT systems.

To find susceptible functions inside the firmware, Binary Code Similarity Detection (BCSD) techniques retrieve functions and compare them with entries in a Common Vulnerabilities and Exposures (CVE) database[2] based on their similarity score. Reverse analysts obtain vital information such as

source code and symbol tables from the BCSD approach, which can be used to identify weaknesses in huge firmware[3]. Vulnerability identification in the IoT sector is crucial for safeguarding against the myriad of security threats [4], [5], [6], [7], [8] caused by the increasing quantity and diversity of connected devices. The major purpose of vulnerability detection in the IoT sector is to protect the security and integrity of IoT networks and devices.

The current approaches to vulnerability detection often fall short in dynamic and heterogeneous IoT environments, where devices generate vast amounts of diverse data. Detecting vulnerabilities in IoT firmware presents several challenges. First, the source code for many IoT firmware images is inaccessible, as they are provided only as binary files, which complicates the security analysis. Second, the diverse array of instruction set architectures (ISAs) on which IoT firmware operates demands extensive reverse engineering expertise and specialized knowledge to identify vulnerabilities effectively. Moreover, the sheer scale of the IoT ecosystem, encompassing over 29 billion devices globally [9], imposes a substantial burden on researchers striving to analyze and discover vulnerabilities comprehensively.

There are notable research gaps in the current vulnerability detection methods for several security threat types, such as command injection, code execution, function level access control, use-after-free, cross-site scripting (XSS), and denial-of-service (DoS). The common weakness enumeration (CWE) [1] list is an industry-standard list of software and hardware weaknesses that pose risks to cybersecurity. For XSS vulnerabilities, there is a notable deficiency in dynamic content analysis and context-aware detection, which results in many false positives and negatives. Command injection vulnerabilities suffer from inadequate comprehensive input validation and a lack of adaptive detection mechanisms that can keep pace with evolving attack techniques. In the realm of Code Execution, the limited capabilities in behavioral analysis and poor integration of static and dynamic analysis methods hinder the effective identification of complex vulnerabilities. Function level access control vulnerabilities highlight the need for more granular detection methods and better mechanisms to identify anomalies in role-based access systems. Use-After-Free vulnerabilities are challenging to detect because of the insufficient tools for analysing complex memory management schemes and the difficulty of identifying temporal vulnerabilities. Addressing these gaps involves developing advanced, context-aware detection mechanisms that integrate static and dynamic analysis, improve behavioral analysis, and prioritize scalability and real-time responses in evolving threat environments.

The novelty of the proposed approach are as follows:

1. The DeBERTa model is used in the proposed approach to generate contextualized embedding, which is further used for vulnerability detection in IoT firmware binaries. The TBL model can comprehend and capture challenging patterns and contexts in the code.
2. The GAT's ability to capture and leverage the structural relationships embedded within the intricate interactions of the firmware's components is also utilized in the proposed approach.
3. Proposed Approach is evaluated on 18 real-world vulnerabilities to show that vulnerable functions are identified with 97.3% accuracy.

The outline of the research paper is as follows: Section 2 narrates the recent works on vulnerability detection. Section 3 details the specifics of the proposed approach. The research findings achieved by the proposed method and its comparative analytics with other approaches are reviewed in Section 4 and the last section concludes with highlighting the novelty of the work proposed.

2. Literature survey

The existing literatures on vulnerability detection in firmware are covered in this section, with a focus on the techniques and frameworks that have been applied to address problems related to firmware security.

Shouguo Yang et al. [10] proposed the tool Asteria-Pro, which combines domain knowledge integration with function encoding. The model effectively removes dissimilar functions using pre-filtration module, thereby reducing computational requirements. Asteria-Pro effectively detects inlined vulnerable functions in more IoT firmware binaries. The Context and Multi-Features-based Vulnerability Detection (CMFVD) framework explained in [11] addresses vulnerabilities in software projects by integrating graphs of code snippets and textual sequences through a slicing technique called context slicing. Zhenhao Luo et al. [3] discussed an approach i.e. IoTSim which is designed using a base-token prediction task for extracting semantics and the Transformer-Based (TB) language framework with disentangled attention to obtain instruction position information. A multi-layer Graph Convolutional Networks (GCN) is used in the approach to generate function embeddings. The IoTSim tool is robust

against detecting similarity in IoT binaries designed from different instruction architecture sets. Issues with IoT firmware images created by compilers with varying degrees of optimization and resulting from diverse architectures are addressed in the work of VulHawk [12]. The VulHawk approach uses the RoBERTa and GCN techniques for generating function embeddings. VulHawk performs 1-day vulnerability detection from IoT Firmware and achieves high performance in detecting vulnerabilities.

The Shouguo Yang et al. [13], aims to decrease false-positive rates and increase accuracy in patch detection across compiler optimization levels and cross-compilers. The model detects and confirms real-world vulnerabilities of ten different real world software programs. In [14], a supervised deep learning approach using recurrent neural networks (RNNs) for vulnerability detection based on binary executables was employed. Differences in the detection of various vulnerabilities were noted, with non-vulnerable samples being identified with a particularly high precision of over 98%. An interaction-based IoT binary similarity comparison system is discussed in [15] . The system uses Bidirectional long short-term memory (Bi-LSTM) and co-attention mechanism, which provides security against IoT malwares.

The method proposed in [16] addresses the challenges in discovering vulnerabilities in SOHO (small office/home office) routers, particularly in their web server modules. DeepWukong [17] represented a drastic improvement in the static detection of software vulnerabilities by integrating deep learning techniques, specifically graph neural networks, into the analysis process. This approach allows for a more nuanced understanding of code, capturing both its logical structure and natural language elements.

Y. Xu et al. [18] represents a significant advancement in binary-level function matching by focusing on patch-based vulnerability identification. The approach focuses on the potential of patch-based approaches in overcoming the limitations of traditional function matching methods, particularly in reducing false positives and improving detection accuracy. The work presented in [19] , uses multiple graph representations, hierarchical convolutional and pooling layers, and a tailored training loss metric for software vulnerability detection. The approach saves time in building static analysers and automates the learning process of insecure patterns from code corpora. In [20], proposed a cross-platform binary vulnerability detection, by leveraging labelled semantic flow graphs, numerical vector extraction, and a customized semantics-aware DNN model. This technique overcomes the limitations of control flow graph (CFG) based methods and improves the detection accuracy up to 83%.

The field of IoT firmware vulnerability detection has seen advancements leveraging static and dynamic analysis, machine learning, and deep learning-based approaches. However, each method has inherent limitations that hinder their comprehensive application, particularly in the diverse and complex domain of IoT firmware security. First, static analysis techniques, such as those used in DeepWukong [17], often result in high false positives because they fail to account for runtime behaviors, making them less effective at detecting vulnerabilities that emerge during execution. Second, tools like Asteria-Pro [10] and VulHawk [12] struggle with analyzing binaries that are heavily obfuscated or optimized, as such conditions obscure the code's structure and semantic details necessary for accurate vulnerability detection. Third, machine learning-based methods, like the supervised RNN approach in [14], depend on extensive labeled datasets, which are challenging to curate for IoT firmware due to the diversity of architectures and the limited availability of real-world vulnerability examples. Finally, methods like IoTSim [3] and the semantic-aware DNN in [20] lack robust adaptability across different instruction set architectures and compiler optimizations, leading to reduced accuracy in cross-platform and cross-compiler scenarios. From the survey, it is also observed that vulnerability type analysis and impact-based detection approaches are lacking in many of the proposed works. Therefore, the proposed model using DeBERTa and GAT technique is introduced. The proposed approach addresses key limitations of existing firmware vulnerability detection methods by incorporating contextual slicing and disentangled attention to retain both static and runtime semantics. The robustness of DeBEERTa, combined with the graph-based modeling capabilities of GAT, effectively handles challenges such as obfuscated or optimized binaries. By overcoming scalability issues in graph-based methods, the proposed method achieves superior accuracy and robustness in detecting vulnerabilities across diverse IoT firmware binaries.

3. Proposed approach

In this work, it is proposed to implement vulnerability detection model for IoT firmware binaries. An overview of the proposed vulnerability detection model using the DeBERTa and GAT

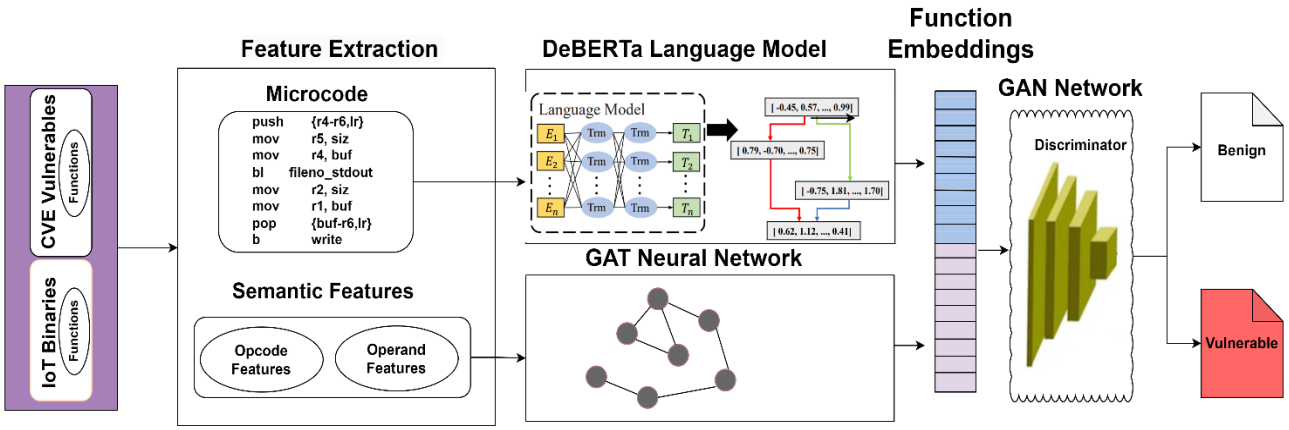

Figure. 1 Overview of the Proposed Approach

technique is shown in Fig. 1. The first step involves extracting relevant features from the IoT firmware and vulnerable binaries. This includes analyzing the microcode, which is a low-level representation of the binary instructions, to derive meaningful attributes. Additionally, features such as opcodes and operands are extracted. These features capture the functional and operational aspects of the binary code.

Preprocessing and Feature Extraction: In the preprocessing stage, the IoT and vulnerable binaries are first disassembled into assembly language using the Binary Ninja tool, thereby converting the binary code into a human-readable format i.e microcode. Next, function boundaries are identified by detecting patterns that indicate the start and end of functions, often marked by prologue and epilogue instructions. Functions are then extracted from the assembly instructions for feature extraction. In the feature extraction stage, features are derived from IoT binary functions and CVE vulnerablilty functions. Each instruction of the function was analyzed to extract operand and opcode features, which provide valuable insights into the structure and behavior of the code. The CPU architecture and instruction set can have an impact on opcode selection. To enhance the understanding of the semantic information in assembly instructions, two feature extraction layers are established: one for opcodes and one for operands. For opcodes, a lookup table is created that is specific to each architecture and converts the opcode type of the input instruction into a one-hot encoded representation[21]. An embedding layer process the vector to produce the opcode-based feature vector, Xopcode. Operands provide the necessary data for the instructions to act upon. The operand features are extracted from the instructions: No. of string literals, no. of integer literals, no. of function names, no. of symbol constants and register sequence features. The

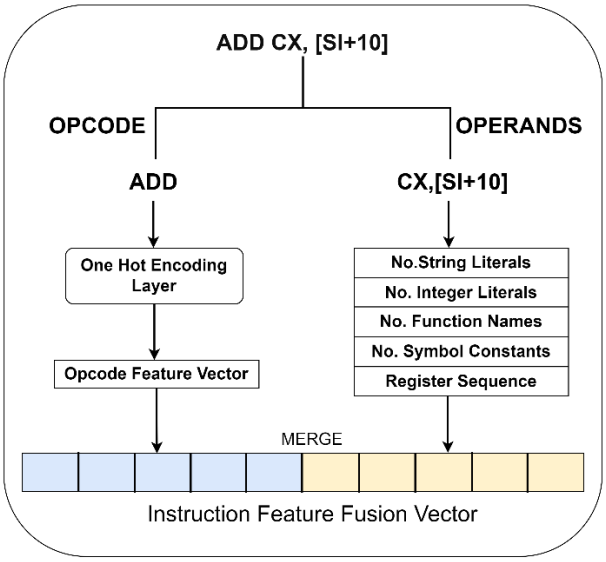

Figure. 2 Instruction feature fusion representation module

overall process of Instruction Feature Fusion Representation Module is depicted in Fig. 2. For Example: opcode-based X_{opcode} , and operands-based $X_{operands}$ feature vector of Add instruction are merged to form the final output X as shown in Eq. (1). These vectorized instruction representations are then utilized as inputs for each node in the graph used for GAT.

$$X = [X_{opcode}; X_{operands}] \quad (1)$$

This concatenated feature vector is used as input to the graph attention network (GAT) neural network for embedding generation. The instructions in microcode are used as input to the DeBERTa model.

DeBERTa Embedding Model: The input functions of IoT firmware and CVE Vulnerables, which consists of instructions, undergo tokenization using

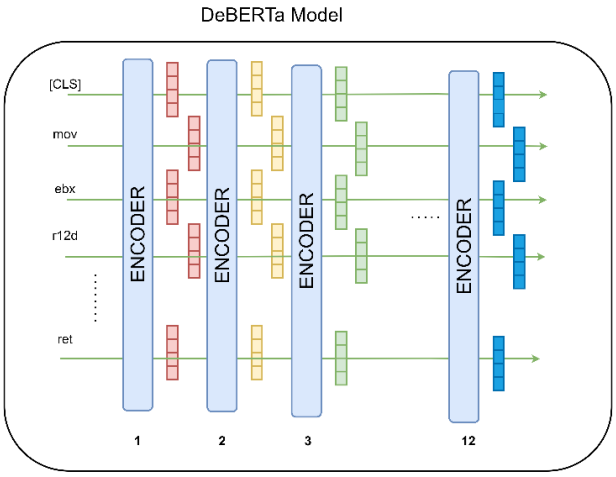

Figure. 3 DeBERTa model for Contextualized Embedding

Byte-Pair Encoding (BPE) technique [22]. This process breaks down the text into subword tokens, making it manageable for the model. Each token then receives an initial word embedding. These embeddings are learned during the model's pretraining phase and encode semantic information on the basis of the token's context in a large corpus of text, typically ranging from tens of gigabytes to several terabytes of text data. To capture the sequential nature of instructions, DeBERTa applies positional encoding to the word embeddings. This step helps the model differentiate tokens based on token position within the input sequence. The core of DeBERTa's architecture lies in its self-attention mechanism. This mechanism allows the model to weigh the importance of each token relative to every other token in the sequence. It enables DeBERTa to capture dependencies and relationships between tokens effectively, which is crucial for understanding complex instructions. The DeBERTa model is represented in Fig. 3, which is employed for generating contextualized embeddings using an encoder architecture. Instructions then pass through multiple transformer layers. In each layer, the model refines the representation of the input sequence by aggregating information across tokens using self-attention and feedforward neural networks. The output from the DeBERTa model, after processing through multiple layers, represents the instruction contextualized embeddings. These embeddings are high-dimensional vectors that encode comprehensive semantic and contextual information about the input instructions. Algorithm 1 presents the embedding computation using the DeBERTa model, that performs the following operations:

Stage 1: Input Representation (Tokenization and Embedding)

In this stage, DeBERTa tokenizes the input sequence into subword tokens and converts them into embeddings. The two types of embeddings are Word Embeddings (E_w) and Positional Embeddings (E_r). Word Embeddings represents the semantic meaning of each token. Positional Embeddings represents the position of each token in the sequence.

Stage 2: Disentangled Attention Mechanism

DeBERTa introduces a disentangled attention mechanism by computing attention based on both content(semantic) and relative position.

A. Content-Based Attention: Attention is calculated by the dot product of query *Q*, key *K*, and value *V* matrices
B. Position-Based Attention: Similarly, positional attention is computed.

The total attention score is then computed by combining content-based and position-based attention:

Stage 3: Multi-Head Attention

In this stage, multi-head attention is applied to the model. For each head *k*, the attention scores are computed independently, and the outputs are concatenated.

Stage 4: Feedforward Network and Residual Connections

After multi-head attention, the output goes through a feedforward neural network. The output *Oi* is passed through a linear transformation followed by a ReLU activation. The residual connections are added to the feedforward output:

Stage 5: Layer Normalization and Output Generation

Layer normalization is applied to stabilize and improve training. The final contextualized embeddings are stored for each token in the sequence. These embeddings capture both the semantic meaning of the words and their contextual relationships.

GAT Embedding Model: Generating instruction embeddings using a graph attention network (GAT) model involves a series of steps that leverage the model's ability to handle graph-structured data and learn from node features and their relationships. Initially, instructions of IoT firmware and vulnerable functions are represented as a graph, where nodes correspond to instructions, and edges denote the relationships between the nodes. Each node is initialized with a feature vector derived from instruction components in the proposed work. No. of string literals, no. of integer literals, no. of function names, no. of symbol constants and register sequence

features are considered. These feature vectors serve as the input embeddings for the GAT model. The GAT model uses an attention mechanism to compute attention coefficients for each pair of connected nodes, determining the importance of each neighboring node's features when updating a node's representation. During the message passing phase, each node aggregates information from its neighbors on the basis of the attention coefficients, combining the aggregated information with its own features to update its representation. Multihead attention is also employed, where multiple attention mechanisms operate in parallel, each providing different perspectives on node relationships. The outputs of multiple heads are concatenated to form the final node representations. Multiple layers of the GAT can be stacked to capture higher-order relationships in the graph, with each layer further refining the node representations. After passing through these layers, the final node representations (embeddings) capture rich semantic and structural information about the instructions. These embeddings can be used for classification tasks. A pooling operation is applied to combine the node embeddings into a single embedding representing the entire instructions of binaries, using the global sum pooling approach [29]. Algorithm 2 presents the embedding computation using GAT model. The steps involved in embedding computation are:

Stage 1: Input Preparation
The input consists of a graph G, Node feature vectors f_i, a weight matrix W and Attention coefficients a.

Stage 2: Feature Transformation
Each node's feature vector is transformed using the weight matrix W, which projects the features into a new space. This transformation ensures that the node features are in a consistent format for further processing.

Stage 3: Attention Score Computation
For each edge (i,j), where j is a neighbor of node i ($j \in N_i$):

A. Concatenate the transformed feature vectors of nodes i and j.
B. Compute the raw attention score m_{ij} using the attention coefficient vector a and the LeakyReLU activation.

Stage 4: Attention Normalization
For each node i, the raw attention scores m_{ij} are normalized using the softmax function to ensure they sum to 1. This produces the final attention coefficient α_{ij}. This normalization ensures that the attention distribution over neighboring nodes is probabilistic.

Stage 5: Feature Aggregation
Using the computed attention coefficients, each node aggregates the feature representations of its neighbors. This process incorporates information from neighboring nodes, weighted by their importance.

Stage 6: Non-Linear Activation
The aggregated features fi' are passed through a ReLU activation function to introduce non-linearity. This step ensures the model can capture complex relationships in the graph.

Stage 7: Graph-Level Embedding (Sum Pooling)
After processing all nodes, the sum pooling operation computes the final graph embedding $\phi(G)$ by summing up the updated feature vectors of all nodes. This produces a single embedding vector that represents the entire graph.

Detection model using GAN: The outputs from the DeBERTa model (semantic and contextual embeddings) and the GAT model (graph embeddings) are concatenated. This step combines the strengths of both models, creating a comprehensive representation that integrates the semantic, contextual, and structural features of the binary code. The concatenated embeddings provide a holistic view of the input data, capturing both the detailed meaning of individual instructions and their relationships within the code. The merged embedding, which now contains integrated features from both DeBERTa and GAT, is then fed into a GAN discriminator.[23, 24] The discriminator is a specialized component designed to analyze the combined embedding and classify the input binaries. By leveraging the detailed and multifaceted representation provided by the merged embedding, the GAN discriminator can differentiate between vulnerable and normal binaries. The GAN discriminator processes the merged embedding and outputs a classification result. This result indicates whether the input binary is classified as vulnerable or normal.

ALGORITHM 1: EMBEDDING COMPUTATION USING DEBERTA

Input: *-Instruction Sequence : $\{I_1,I_2,I_3,....I_n\}$*
-Projection matrix: W
Output *Contextualized Embedding:*

1. *ContextEmb* $\leftarrow \phi$
2. ***for each*** *Instruction s* $\in \{I_1,I_2,I_3,....I_n\}$
3. $Input_{tokens} \leftarrow Tokenizer\ (s)$

for each token i ∈ $Input_{tokens}$
  Compute word embeddings:
    $E_w \leftarrow$ Token_Embedding(i)
  Compute relative position embeddings:
    $E_r \leftarrow$ Positional_Embedding(i)
  for each head k ← {1,2, 3,.. 12}
    Compute Content Based Attention Scores:
      $Q_{w_k}, K_{w_k}, V_{w_k} = E_w \times W_{w_k}^Q, \quad E_w \times W_{w_k}^K, \quad E_w \times W_{w_k}^V$
    Compute Position Based Attention Scores:
      $Q_{r_k}, K_{r_k} = \quad E_r \times W_{r_k}^Q, \quad E_r \times W_{r_k}^K$
    Compute the disentangled attention scores:
      $Attention(Q_{w_k}, K_{w_k}, Q_{r_k}, K_{r_k}) = Softmax\left(\frac{Q_{w_k}K_{w_k}^T + Q_{r_k}K_{r_k}^T}{\sqrt{d_k}}\right)$
    For token i , all tokens in the input sequence j, the output of the attention mechanism of k^{th} head is
      $O_k = \sum_j Attention_{ij} \cdot V_{w_k}$
  end
  Concatenate the outputs from all heads of token i:
    $O_i \leftarrow Concat(O_k^1, O_k^2, \dots, O_k^{12}) . W_O$
  Apply a feedforward neural network with ReLU activation:
    $F_i = ReLU(O_i W_1 + b_1)W_2 + b_2$
  $X_i \leftarrow$ Add residual connections and apply layer normalization to the output F_i
  ContextEmb ← ContextEmb U { X_i }
end
end
return ContextEmb

ALGORITHM 2: EMBEDDING COMPUTATION USING GAT MODEL

Input: - Graph G : Nodes(n) and Edges(e)
- Node feature Vector: f, Weight Matrix: W
- Attention Coefficients: a

Output: Graph Embedding: $\phi\ (G)$

Apply the weight matrix W to each node feature vector $\vec{f}_i$
  $\vec{f}'_i = \boldsymbol{W}.\vec{f}_i \quad \forall i \in \{1,\dots,n\}$
for each edge (i, j) where j ∈ $\mathcal{N}eighbours(N_i)$
  Concatenate the transformed feature vectors of nodes i and j
    $\vec{f}_{ij} = \vec{f}'_i || \vec{f}'_j$
  Compute the raw attention score using the weight vector $\vec{a}$ and LeakyReLU activation
    $m_{ij} = LeakyReLU(\vec{a}^\top . \vec{f}_{ij})$
end
for each vertex i
  Normalize the attention scores using the softmax function
    $\alpha_{ij} = \frac{exp(m_{ij})}{\sum_{k \in N_i} \exp(m_{ik})}$
  Aggregate the transformed features of its neighbors weighted by the attention coefficients:
    $f_i'' = \sum_{j \in N_i} \alpha_{ij} . \vec{f}'_j$
  Apply a non-linear activation function to the aggregated features:
    $f_i''' = ReLU(f_i'')$
end
Compute the sum of the updated feature vectors (Sum Pooling):
  $\phi\ (G) = \Sigma_{i=1}^n f_i'''$
return $\phi(G)$

Table 1. Variable and its definition

Variable	Definition
I_n	Instruction Sequence
K	Attention Head count
n	Number of nodes in Graph
e	Number of edges in Graph
W	Projection matrix
E_w	Word Embedding
E_r	Relative positional embedding
$Q_{w_k}, K_{w_k}, V_{w_k}$	Content-based attention scores
ContextEmb	Contextualized Embedding
a	Attention Coefficient
$\vec{f}_i$	Feature Vector
α_{ij}	Normalized Attention Score
$\phi\ (G)$	Graph Embedding
Acc	Accuracy
Comp	Compatible

4. Experimental results

This section presents the results and a performance assessment of the proposed vulnerability detection model. The Proposed approach is compared with the latest works for

detecting vulnerabilities in IoT firmware binaries. The implementation language used in the proposed approach is Python v3.12.5, with the PyTorch framework as the primary tool. For disassembly, Binary Ninja is utilized. The server operating system is Ubuntu 22.04 LTS, supported by a 12-core Intel Xeon E5-2697v2 CPU running at 2.7 GHz, and 128 GB of server memory. The GPU in use is an NVIDIA RTX 3090. The DeBERTa model used in this setup consists of twelve layers with output embeddings having a dimension of 256. The function embedding model is implemented with four-layer Graph Attention Networks (GATs) and uses binary cross-entropy as the function. Training and evaluation are conducted on a desktop computer running Windows 10.

Experiments are conducted separately for each of the 18 types of vulnerabilities. The training method follows a batch-wise approach, with a batch size of 64, and each node has a vector dimension of 64. The dropout rate is set at 0.5, and the number of epochs ranges from 10 to 100. Optimization is performed using the Adam optimizer, with a learning rate of 0.002. The dataset presented provides a comprehensive view of firmware analysis. In this experiment, the latest IoT firmware images from nine vendors (Tasmota, OpenWrt, MicroPython, Contiki-NG, RIOT-OS, Cisco, D-Link, TP-Link and Netgear) are collected. The custom CVE vulnerability repository is built for the current experiment, which is based on the CVE database. The dataset is based on IoT firmware and associated vulnerabilities. The CVE vulnerabilities[34] considered in the proposed approach are listed in Table 2. The dataset includes 49 firmware images from nine vendors, as documented in Table 3, and 230 vulnerable functions from IoT projects, as detailed in Table 4. In total, the dataset comprises 113,508 functions collected from nine different IoT projects, including 230 vulnerable functions. The vulnerabilities include Memory

Table 2. Key Vulnerabilities used in Dataset

Vulnerabilities	CVE Ids	Description
CWE-787	CVE-2022-43294, CVE-2023-28116, CVE-2023-23609, CVE-2018-16666, CVE-2023-24817, CVE-2023-24797, CVE-2024-22751	Out-of-bounds Write
CWE-79	CVE-2021-36603, CVE-2023-24182, CVE-2019-18993	Cross-site Scripting
CWE-122	CVE-2023-7158	Heap-based Buffer Overflow
CWE-119	CVE-2018-16665, CVE-2023-33975	Improper Restriction of Operations within the Bounds of a Memory Buffer
CWE-20	CVE-2021-44228	Improper Input Validation
CWE-77	CVE-2022-25060, CVE-2022-25064, CVE- 2022-27647	Command Injection
CWE-120	CVE-2022-27643	Classic Buffer Overflow

Table 3. Firmware Binaries used in Dataset

IoT Firmwares	# Firmware Images
Tasmota[25]	3
OpenWrt [26]	4
MicroPython[27]	2
Contiki-NG[28]	5
RIOT-OS[29]	2
Cisco[30]	3
D-Link[31]	10
Tp-Link[32]	10
Netgear[33]	10
TOTAL	49

Table 4. Vulnerabilities in Different Versions of Firmware

CVE Ids	Function Name (Filename)	Confirmed#
2022-43294	ClientPortPtr (CRtspSession.cpp)	8
2021-36603	ble-l2cap (ble-l2cap.c)	11
2023-7158	slice_indices (objslice.c)	12
2023-28116	ble-l2cap (ble-l2cap.c)	17
2023-23609	ble-l2cap (ble-l2cap.c)	16
2018-16666	next_string (aql-lexer.c)	17
2018-16665	lvm_shift_for_operator (lvm.c)	20
2023-33975	rbuf_add	13
2023-24817	gnrc_rpl_srh_process	14
2021-44228	log4j-core	12
2024-22751	sub_477AA0	11
2023-24797	sub_48AC20	8
2022-25060	oal_startPing	17
2022-25064	oal_wan6_setIpAddr	16
2022-27647	Libreadycloud	12
2023-24182	sshkeys	7
2019-18993	New_port_forward	12
2022-27643	SOAPAction_header,	7

Table 5. X-Opt Vulnerability detection accuracy of proposed approach

Firmware	Tasmota				OpenWrt				Micro Python				Contiki-NG			
	Opt0	Opt1	Opt2	Opt3	Opt0	Opt1	Opt2	Opt3	Opt0	Opt1	Opt2	Opt3	Opt0	Opt1	Opt2	Opt3
Opt0	**0.97**	0.79	0.85	0.89	**0.95**	0.85	0.78	0.99	**0.88**	0.85	0.89	0.95	**0.94**	0.77	0.99	0.75
Opt1	0.88	**0.98**	0.94	0.85	0.87	**0.94**	0.84	0.85	0.87	**0.99**	0.66	0.85	0.93	**0.93**	0.85	0.87
Opt2	0.87	0.88	**0.97**	0.79	0.77	0.86	**0.93**	0.82	0.88	0.22	**0.95**	0.88	0.83	0.86	**0.92**	0.86
Opt3	0.85	0.89	0.91	**0.94**	0.85	0.89	0.91	**0.96**	0.85	0.87	0.91	**0.92**	0.89	0.90	0.90	**0.96**
Firmware	Cisco				D-Link				Tp-Link				Netgear			
	Opt0	Opt1	Opt2	Opt3	Opt0	Opt1	Opt2	Opt3	Opt0	Opt1	Opt2	Opt3	Opt0	Opt1	Opt2	Opt3
Opt0	**0.98**	0.84	0.82	0.89	**0.91**	0.85	0.86	0.84	**0.96**	0.81	0.85	0.86	**0.95**	0.79	0.81	0.83
Opt1	0.83	**0.89**	0.89	0.85	0.83	**0.90**	0.84	0.85	0.81	**0.92**	0.84	0.87	0.86	**0.96**	0.84	0.85
Opt2	0.87	0.77	**0.88**	0.89	0.85	0.88	**0.91**	0.80	0.85	0.96	**0.97**	0.84	0.87	0.96	**0.90**	0.89
Opt3	0.79	0.78	0.91	**0.91**	0.84	0.89	0.91	**0.87**	0.85	0.87	0.91	**0.98**	0.85	0.99	0.85	**0.91**

Corruption, XSS, Buffer Overflow (Heap-based), Out-of-Bounds Memory Access, Improper Input Validation, Command Injection, and Classic Buffer Overflow. Cross Optimization (X-Opt) analysis in vulnerability detection helps in understanding the effect of optimizations in presence and behavior of vulnerabilities, ensuring the effectiveness of security tools, and developing robust security strategies. To reflect real-world scenarios, the binaries were built using default compilation settings without any additional compiler optimizations. Each program in binaries was compiled at four different optimization levels (Opt0, Opt1, Opt2, Opt3). Opt0 indicates that no optimization is performed. Opt1 indicates the restricted level of optimizations performed. Opt2 represented a high level of optimization performed and Opt3 refers to the most aggressive level of optimization. The programs are compiled at the Opt0 optimization degree for the cross-compiler dataset via GCC (version 2022.1) and Clang (version 7.0).

In X-Opt detection, the proposed strategy is applied to analyze firmware binaries compiled with separate optimizations levels. Table 5 displays the X-Opt level vulnerability detection accuracy for IoT firmware packages. The columns labelled Opt0 present the non-X-Opt detection results, whereas the columns labelled Opt1, Opt2, Opt3 display the cross-optimization detection results. The top row indicates the names of the IoT firmware binaries used for the experiment. The next rows indicate the optimization degrees applied to compile the target firmware, whereas the columns indicate the optimization degrees applied to compile the vulnerability. The cross-optimization accuracy of each firmware helps in analysing how different compiler optimizations impact the performance, size, and potentially the presence of vulnerabilities in the compiled software. In Table 5, the detection accuracy in non-X-Opt settings is represented by the bold numbers, which serve as a baseline for the findings achieved in X-Opt settings. High accuracy is achieved by proposed method at all levels of optimization, ranging from 75% to 99%, with an average accuracy of 88.0%. Specifically, higher accuracy is obtained when the target and vulnerablities compiled at the same optimization level (87% - 98%) compared to different optimization levels (75% - 89%).

Table 6. Analysis of CVE Function Sizes and Their Corresponding Detection Accuracy, runtime, and memory usage in different Test Cases

Function Size	**# Target Test Functions**	**Detection Acc**	**Memory Usage (MB)**	**Average Runtime (s)**
1K-20K	20	96.7%	128	2.4
21K-40K	26	98.3%	240	5.8
41K-60K	35	97.5%	412	10.9
61K-80K	28	96.3%	654	19.3
81K-1M	44	97.7%	800	30
>1M	27	96.9%	936	45
***K is short for Kilo, M is short for Million**				

Accuracy of functions of different sizes: A statistical analysis was conducted on the distribution of function sizes (i.e., total number of pointers, method calls, and objects) across 180 tested functions and their vulnerability detection accuracy. Table 6 presents the detection accuracy, memory usage and average runtime of the target test functions on the basis of their function size. The first column lists the ranges of function sizes. The second column indicates the number of functions falling within each boundary range. The function sizes range from 1K to >1M, with the majority being less than 1M. The accuracy of vulnerability detection for functions of different sizes is shown in the third column. The accuracy of vulnerability detection ranged from 0.963 to 0.983, suggesting reliable detection without a noticeable trend of decreasing accuracy as function size increased. The system has been tested across a wide range of function sizes, from small (1K-20K) to extra-large (>1M).

The average detection accuracy across all tested functions is 97.3%, indicating a high level of reliability in vulnerability detection. The highest detection accuracy is observed in the 21K-40K size range, whereas the lowest is in the 60K-80K size range. Despite variations in accuracy, the system maintains a consistently high detection rate above 96% across all function sizes. The impact of the total number of pointers, method calls, and objects on vulnerability detection is reflected in the detection accuracy across different function sizes. The small functions (1K-20K) with fewer pointers, method calls, and objects have a detection accuracy of 96.7%. The medium-sized functions (21K-40K and 41K-60K) achieve the highest detection accuracies, 98.3% and 97.5% respectively, as the system effectively identifies complex patterns. Large functions (61K-80K and 81K-1M) exhibit a slight decrease in accuracy to 96.3% but rebound to 97.7% because of the system's robustness.

Extra-large functions (>1M), with the highest number of pointers, method calls, and objects, maintain a strong detection accuracy of 96.9%. One major problem with having more pointers, method calls, and objects in vulnerability detection is the increased complexity of the codebase. This complexity makes it harder to analyse and track the flow of data and control within the program, potentially leading to increased chances of missing vulnerabilities. Pointers can introduce issues such as dangling pointers, buffer overflows, and memory leaks. Increased method calls can obscure the logical flow and create intricate dependencies that are difficult to trace. A greater number of objects can lead to complicated interactions and state management issues. As the number of pointers, objects and calls increases in function, more complex interactions occur. Despite increasing complexity, the system demonstrates a consistently high average detection accuracy of 97.3% across all function sizes, indicating its reliability and effectiveness in vulnerability detection. Memory requirements increase proportionally with function size due to the larger feature set and computations involved. The memory usage grows from 128 MB for small functions (1K–20K) to 936 MB for large functions (>1M), highlighting the scalability of the system. While the runtime increases with function size, the system maintains reasonable processing times. Small functions are analyzed in just 2.4 seconds, whereas very large functions (>1M) require 45 seconds, which is acceptable for detailed vulnerability analysis. The model was tested against adversarially perturbed firmware binaries created by introducing modifications, such as opcode substitutions, instruction reordering, and the addition of no-operation (NOP) instructions. The results are shown in Table 7.

Table 7. Adversarial Testing Results on Firmware Binaries

Attack Type	Accuracy	Comments
Proposed Approach	97.50%	Normal binaries without adversarial perturbations were used.
Opcode Substitution	87.20%	Minor degradation observed due to substitution of opcodes with semantically similar ones.
Instruction Reordering	84.80%	Reordering instructions caused a moderate drop in performance.
NOP Padding	85.30%	Extra NOP instructions slightly affected feature alignment and detection.
Combination of All	80.10%	Significant degradation observed under combined attacks.

4.1 Baseline approaches

For an exhaustive comparison, the evaluation includes the following cutting-edge approaches selected as baselines.

The benchmark approaches are as follows:

- IoTSim [3], an IoT-oriented BCSD tool that integrates TBL models with disentangled attention and a multilayer GCN to detect vulnerabilities in IoT firmware binaries.

- DeepWukong [17] used a novel DL-based embedding technique for static detection of software weakness in C/C++ programs, using advanced GNN to encode code snippets.
- Robin [13], proposed an approach to enhance binary code similarity detection by filtering out false positives caused by vulnerable and patched functions.

4.2 Performance metrics

To evaluate the effectiveness of proposed model in vulnerability detection, four performance metrics, including Accuracy, Precision, Recall, and F1 score are used. Accuracy measures the overall correctness of vulnerability detection results. Precision denotes the proportion of identified vulnerabilities that are true positives among all identified vulnerabilities. Recall measures the proportion of true vulnerabilities correctly identified by proposed approach. The F1 score, which combines precision and recall, provides a single metric to assess proposed approach ability to accurately detect and distinguish between vulnerabilities and patched functions across various compiler settings and software types.

Table 8. Comparison of Cross Compiler Vulnerability Detection Results

Compiler	GCC		Clang	
	Comp#	Acc	Comp#	Acc
IoTSim[3]	170	78.2%	158	65.5%
DeepWukong [17]	160	81.4%	141	72.6%
Robin[13]	175	84.2%	149	73.5%
Proposed Approach	**180**	**89%**	**180**	**88%**

Related Works Comparison: Two firmwares (D-Link and Tp-Link) with the highest number of test instances were selected for cross-compiler analysis [13,35], and firmware was compiled via GCC (version 2022.1), and Clang (version 7) compilers. Two separate compilers are used to determine the cross-compiler detection accuracy. The compiler optimization level is set to Opt0 during compilation.

Table 8 lists the test cases that are compatible with the experiments, along with the current works and proposed approach accuracy for cross-compiler vulnerability detection. The detection outcomes of the proposed approach, are displayed bold. The proposed method achieves 88% and 89% accuracy on

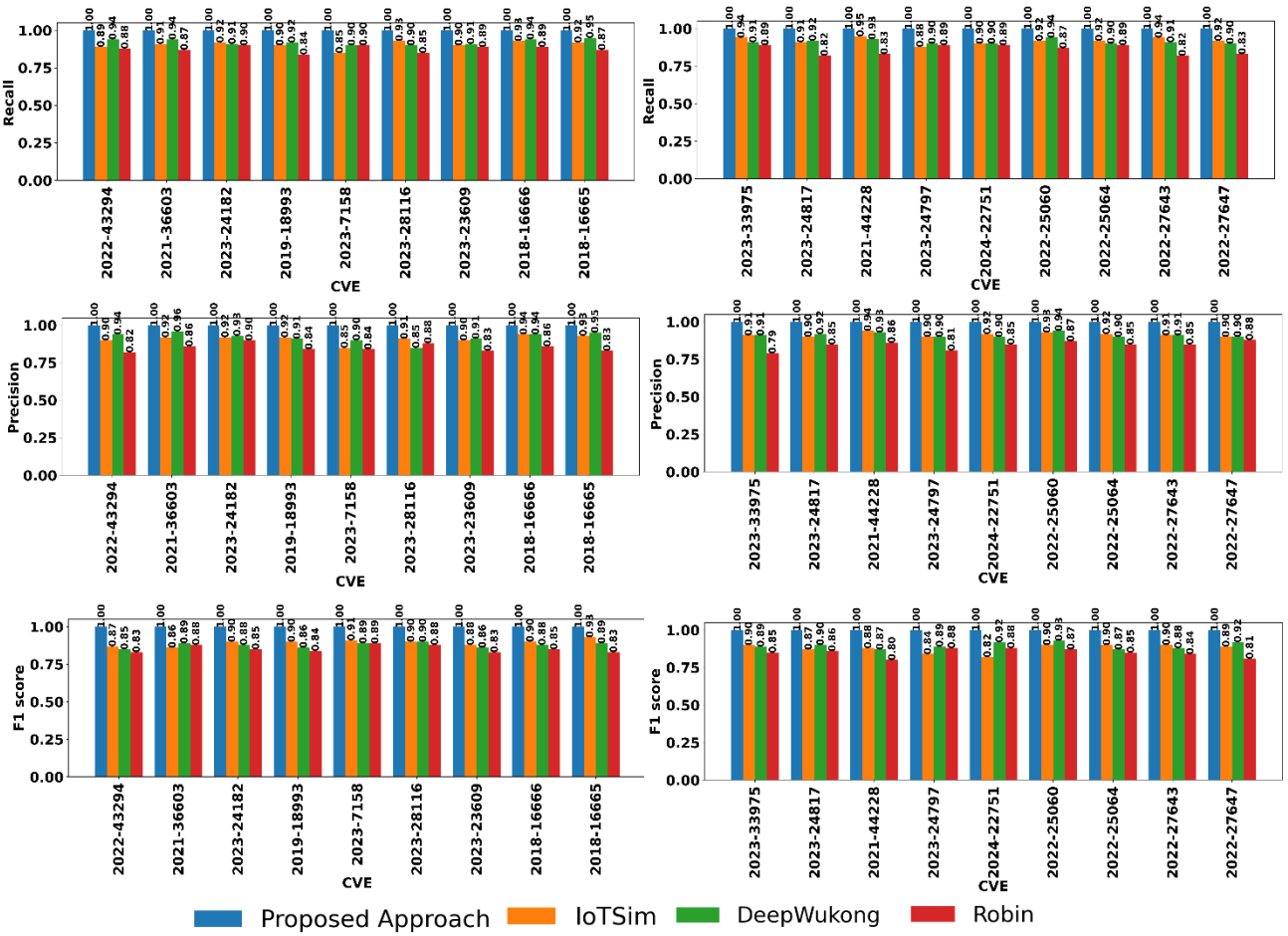

Figure. 4 Recall, Precision, F1-score results of vulnerability detection in IoT Binaries

binaries that are compiled with Clang and GCC compilers, respectively. The detection accuracy in Clang is marginally lower than that in GCC because the Clang compiler uses different stacks and registers in binaries. The detection accuracy is low in [3] and [17] because Clang compiler lacks support for older language standards. [13] has low detection accuracy and high scalability because of syntactic changes in the code. The results indicate, that the proposed model is used as a reference model when comparing the outcomes of other tools. Fig. 4 presents the outcomes of firmware vulnerability detection, showcasing the recall, precision and F1 score respectively. When comparing proposed method with other current approaches, it is evident that for most of the CVEs, the performance of proposed approach is significantly better (by average >8%) than that of current approaches such as IoTSim, DeepWukong, and Robin.For example, in the cases of CVE-2023-28116, CVE-2023-23609, CVE-2018-16666 and CVE-2018-16665 from the Contiki-NG project, proposed method achieves a recall of 100%. The proposed approach is successful in identifying all 70 vulnerable functions of the Contiki-NG project. The average recall values achieved by IoTSim, DeepWukong, and Robin are 92%, 92.5%, and 87.5%, respectively. The high recall achieved by the proposed method in vulnerability detection indicates that the proposed approach is highly effective at identifying true vulnerabilities within the dataset. The research outcomes indicate that the suggested approach can be used to identify vulnerabilities in IoT scenarios in an efficient and dependable manner.

4.3 Discussions

The ability of the proposed method is evident in the detection of firmware binaries vulnerabilities. However, it's important to take into account certain limitations. One key concern is the potential biases introduced during dataset preprocessing, such as tokenization, graph construction, and feature selection, which may impact the model's ability to generalize across diverse datasets. Scalability poses another challenge, as the method may struggle to handle large-scale datasets or complex graphs due to increased computational and memory requirements. Additionally, the performance of the DeBERTa and GAT models is highly sensitive to hyperparameter settings, and suboptimal configurations could lead to reduced accuracy or instability. The approach also faces limitations in generalizing across diverse instruction architectures, potentially hindering its effectiveness in specific edge cases. Lastly, the interpretability of the model remains a concern, as it may be difficult to understand how specific predictions are made, limiting the ability to diagnose errors or explain outcomes effectively.

5. Conclusion and future work

In this work, a vulnerability detection tool uses a GAT network and DeBERTa model, is proposed. By leveraging DeBERTa's disentangled attention mechanism and robust semantic embedding capabilities, the proposed approach can achieve a deeper understanding of the code context and semantics, resulting in more accurate and effective identification of vulnerabilities. The GAT improves vulnerability detection by effectively capturing code structure and interdependencies through attention mechanisms, enhancing accuracy and scalability in analysing large codebases. Under cross-compiler settings, the proposed method achieves superior performance with 89% accuracy across GCC and Clang compiler, significantly outperforming cutting-edge programs like Robin, DeepWukong and IoTSim highlighting its robustness and effectiveness in diverse environments. High accuracy is achieved by the proposed method at all levels of optimization, with an average accuracy of 88.0% across all firmware. The system consistently achieves a high average detection accuracy of 97.3% across all function sizes, demonstrating its reliability in vulnerability detection. The evaluation results show a recall of 100%, successfully identifying all 18 vulnerability types. Future work should focus on improving dataset preprocessing techniques to mitigate biases introduced during tokenization, enhancing the model's generalization across diverse datasets. Additionally, efforts should be made to enhance model interpretability and adaptability across different firmware types and instruction architectures, ensuring more robust and explainable predictions.

Conflicts of Interest

All authors declare no conflict of interest.

Author Contributions

Conceptualization, methodology, writing original draft preparation, Nandish M and Jalesh Kumar; Supervision, Jalesh Kumar

References

[1] Common Weakness Enumeration, *Software and Hardware weaknesses DB*. Accessed:

Jan. 02, 2023. [Online]. Available: https://cwe.mitre.org/

[2] National Vulnerability Database, *National Institute of Standards and Technology*. Accessed: Apr. 13, 2024. [Online]. Available: https://nvd.nist.gov/

[3] Z. Luo, P. Wang, W. Xie, X. Zhou, and B. Wang, "IoTSim: Internet of Things-Oriented Binary Code Similarity Detection with Multiple Block Relations", *Sensors*, Vol. 23, No. 18, pp.1-22, 2023.

[4] T. Bakhshi, B. Ghita, and I. Kuzminykh, "A Review of IoT Firmware Vulnerabilities and Auditing Techniques", *Sensors*, Vol. 24, pp.1-28, 2024.

[5] M. H G, J. Kumar, and N. M, "GrMA-CNN: Integrating Spatial-Spectral Layers with Modified Attention for Botnet Detection Using Graph Convolution for Securing Networks", *International Journal of Intelligent Engineering and Systems*, Vol. 18, No. 1, pp. 1009-1020, 2025, doi: 10.22266/ijies2025.0229.72.

[6] S. Ul Haq, Y. Singh, A. Sharma, R. Gupta, and D. Gupta, "A survey on IoT & embedded device firmware security: architecture, extraction techniques, and vulnerability analysis frameworks", *Discov Internet Things*, Vol. 3, pp.1-29, 2023, doi: 10.1007/s43926-023-00045-2.

[7] X. Feng, X. Zhu, Q. L. Han, W. Zhou, S. Wen, and Y. Xiang, "Detecting Vulnerability on IoT Device Firmware: A Survey", *IEEE/CAA Journal of Automatica Sinica*, Vol.10, No. 1, pp. 25-41, 2023, doi: 10.1109/JAS.2022.105860

[8] T. Sasi, A. H. Lashkari, R. Lu, P. Xiong, and S. Iqbal, "A comprehensive survey on IoT attacks: Taxonomy, detection mechanisms and challenges", *Journal of Information and Intelligence*, Vol. 2, No. 6, pp.455-513, 2023.

[9] IoT Analytics, *State of IoT*. Accessed: Jan. 01, 2023. [Online]. Available: https://iot-analytics.com/number-connected-iot-devices/

[10] S. Yang *et al.*, "Asteria-Pro: Enhancing Deep Learning-based Binary Code Similarity Detection by Incorporating Domain Knowledge", *ACM Transactions on Software Engineering and Methodology*, Vol. 33, No. 1, pp.1-39, 2023.

[11] Y. Zhang, Y. Hu, and X. Chen, "Context and Multi-Features-Based Vulnerability Detection: A Vulnerability Detection Frame Based on Context Slicing and Multi-Features", *Sensors*, Vol. 24, No. 5, pp.1-21, 2024.

[12] Z. Luo *et al.*, "VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search", In: *Proc. of 30th Annual Network and Distributed System Security Symposium, NDSS 2023*, pp.1-18, 2023.

[13] S. Yang *et al.*, "Towards Practical Binary Code Similarity Detection: Vulnerability Verification via Patch Semantic Analysis", *ACM Transactions on Software Engineering and Methodology*, Vol. 32, No. 6, pp.1-29, 2023.

[14] A. Schaad and D. Binder, "Deep-Learning-based Vulnerability Detection in Binary Executables", *Cryptography and Security*, pp.1-21, 2022.

[15] Q. Song, Y. Zhang, B. Wang, and Y. Chen, "Inter-BIN: Interaction-based Cross-architecture IoT Binary Similarity Comparison", *IEEE Internet of Things Journal*, Vol.9, No. 20, pp.20018-20033, 2022, doi: 10.1109/JIOT.2022.3170927.

[16] Y. Zhang *et al.*, "ESRFuzzer: an enhanced fuzzing framework for physical SOHO router devices to discover multi-Type vulnerabilities", *Cybersecurity*, Vol. 4, No. 1, pp.1-22, 2021.

[17] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network", *ACM Transactions on Software Engineering and Methodology*, Vol. 30, No. 3, pp.1-30, 2020.

[18] Y. Xu, Z. Xu, B. Chen, F. Song, Y. Liu, and T. Liu, "Patch based vulnerability matching for binary programs", In: *Proc. of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 376–387, 2020.

[19] Y. Zhuang, S. Suneja, V. Thost, G. Domeniconi, A. Morari, and J. Laredo, "Software Vulnerability Detection via Deep

Learning over Disaggregated Code Graph Representation", *Artificial Intelligence*, pp.1-11, 2021.

[20] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary", In: *Proc. of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 896–899, 2018.

[21] J. T. Hancock and T. M. Khoshgoftaar, "Survey on categorical data for neural networks", *J Big Data*, Vol. 7, pp. 1-41, 2020, doi: 10.1186/s40537-020-00305-w

[22] V. Zouhar et al., "A Formal Perspective on Byte-Pair Encoding", In: *Proc. of Findings of the Association for Computational Linguistics: ACL 2023*, pp. 598–614, 2023, doi: 10.18653/v1/2023.findings-acl.38.

[23] I. J. Goodfellow *et al.*, *Generative Adversarial Networks*. Jun. 2014, [Online]. Available: http://arxiv.org/abs/1406.2661.

[24] S. Karthika and M. Durgadevi, "Generative Adversarial Network (GAN): a general review on different variants of GAN and applications", In: *Proc. of 2021 6th International Conference on Communication and Electronics Systems*, pp. 1–8, 2021.

[25] IoT Firmware, "Tasmota." Accessed: May 05, 2023. [Online]. Available: https://github.com/arendst/Tasmota-firmware/tree/firmware/firmware

[26] IoT Firmware, "OpenWrt." Accessed: Mar. 04, 2023. [Online]. Available: https://github.com/OWASP/IoTGoat/tree/master/OpenWrt

[27] IoT Firmware, "MicroPython." Accessed: Mar. 03, 2023. [Online]. Available: https://github.com/peterhinch/micropython-iot/tree/master

[28] IoT Firmware, "Contiki-NG." Accessed: Jan. 03, 2023. [Online]. Available: https://github.com/contiki-ng/contiki-ng

[29] [29] IoT Firmware, "RIOT-OS." Accessed: Aug. 07, 2023. [Online]. Available: https://github.com/RIOT-OS/RIOT

[30] IoT Firmware, "CISCO." Accessed: Jul. 08, 2023. [Online]. Available: https://software.cisco.com/download/home

[31] IoT Firmware, "D-Link." Accessed: Feb. 02, 2024. [Online]. Available: https://tsd.dlink.com.tw/ddwn

[32] IoT Firmware, "TP-Link." Accessed: Jan. 01, 2024. [Online]. Available: https://download1.dd-wrt.com/dd-wrtv2/downloads/betas/2023/04-02-2023-r52217/tplink_tl-wr840nv1/

[33] IoT Firmware, "Netgear." Accessed: Jul. 06, 2023. [Online]. Available: https://download1.dd-wrt.com/dd-wrtv2/downloads/betas/2024/01-02-2024-r54682/netgear-r6400/

[34] CWE Weakness, *CWE Top 25 Weakness 2023*. Accessed: Nov. 11, 2023. [Online]. Available: https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html#tableView

[35] Paul Black, Iqbal Gondal, "Cross-Compiler Bipartite Vulnerability Search", *Electronics*, Vol.10, No. 11, pp.1-17, 2021.