DeepWare: Attention-based CNN-BiLSTM for Detecting Malware in Embedded Systems

Journal:	*IEEE Internet of Things Journal*
Manuscript ID	IoT-22608-2022
Manuscript Type:	Regular Article
Date Submitted by the Author:	16-Feb-2022
Complete List of Authors:	Hamad, Salma; Macquarie University, Computing Tran, Dai; Macquarie University, Computing Sheng, Michael Zhang, Wei; University of Adelaide, Computing
Keywords:	IoT Security, Malware Detection, Static Analysis

DeepWare: Attention based CNN-BiLSTM for Detecting Malware in Embedded Systems

Salma Abdalla Hamad*, Dai Hoang Tran*, Quan Z. Sheng*, Wei Emma Zhang†
*School of Computing, Macquarie University, Sydney, NSW 2113, Australia
Email: salma.hamad@mq.edu.au, michael.sheng@mq.edu.au
†School of Computer Science, The University of Adelaide, Adelaide, SA 5005, Australia
Email: wei.e.zhang@adelaide.edu.au

***Abstract*—Detecting malware in embedded systems is becoming a critical task in various Internet of Things (IoT) infrastructures in which embedded devices perform crucial tasks (e.g., monitoring expensive engines). The complexity of this duty increases with the recent growth of malware variants targeting different embedded CPU architectures. Traditionally, machine learning techniques require feature engineering before training, which requires domain knowledge that might not be applicable for unseen malware variants. Recent deep learning approaches have performed well on malware analysis and detection, eliminating the need for the feature engineering phase.**

In this work, we propose DeepWare, a real-time cross-architecture malware detection solution tailored for embedded systems. It detects malware by analyzing the binary file's executable operation codes (OpCodes) sequence representations. In particular, we use Bidirectional Encoder Representations from Transformers (BERT) Embedding, the state-of-the-art natural language processing (NLP) approach to extract contextual information within an executable file's OpCode sequence. The sentence embedding output generated from BERT is fed into a hybrid multi-head CNN-BiLSTM-LocAtt model that realizes the resources of the convolutional neural network (CNN) and bidirectional long short-term memory (BiLSTM) with added benefits from the local attention mechanism. The proposed model extracts the semantic and contextual features and capture long-term dependencies between OpCode sequences, which helps improve the detection performance.

We train and evaluate the proposed DeepWare[1] using the datasets created for three different CPU architectures. Our proposed solution achieves F1-scores of 99.4%, 97.1%, and 94.2% for the ARM, PowerPC, and MIPS datasets, respectively. The performance evaluation results show that the proposed multi-head CNN-BiLSTM-LocAtt model produces more accurate classification results with higher recall and F1-scores than both traditional machine learning classifiers and individual BiLSTM models.

***Index Terms*—Malware Detection, Embedded Devices, Static Analysis**

I. Introduction

The advancement and widespread of modern embedded systems such as Internet of Things (IoT) infrastructures have prompted cyber-criminals to devise dangerous and refined attacks against embedded devices [1], [2]. Adversaries are now increasingly focusing on the misuse of low-level vulnerabilities to infect the target systems. Meanwhile, malware detection and prevention remain active research areas [3], [4]. However, due to the resource-constrained nature of IoT device's hardware and the customized operating systems, the available malware detection methodologies for embedded systems are not tailored for real-world deployment. There is thus an urgent need for specific malware detection of embedded systems.

Traditional signature-based malware detection methods depend on the accumulation of signature libraries and malware analysts' expertise. It is challenging to accommodate embedded malware growth. On the other hand, machine learning techniques and data analysis methods have recently been extensively utilized and deployed to provide an automatic and efficient pipelines for solving malware detection problem [5].

Dynamic and static approaches are the two main approaches for malware analysis and automated detection techniques. Both methods can utilize machine learning techniques to detect malware. The dynamic approaches [6], [7] monitor the execution of binary executables to see abnormal behaviors. However, the run-time monitoring has several common potential weaknesses, such as requiring isolation, substantial resources, and the high overhead in execution run-time [5]. Moreover, executing the malicious binary files could infect the physical environments [8]. Regarding IoT dynamic malware detection, the most inherent disadvantages are due to diverse microcontroller architectures (e.g., MISP, ARM, PowerPC, x86) [9] and resource constraints of IoT devices. Hence, it is not easy to correctly set up the required environment for running and monitoring the IoT executables. On the other hand, the static method analyzes and observes the binaries' structure without executing them. Thus, it is not necessary to build a specific run-time environment for each architecture, and the detection time is usually faster than dynamic malware detection methods. However, static analysis performance can be affected by code obfuscation [6].

Most of the recent IoT developed malware are descendant or evolved variants of Bashlite [10] and Mirai malware [11], [12]. Consequently, IoT malware families have similar implemented source codes and operational functions. Based on such similarities, there are several static features of the IoT malware, such as Executable and Linkable format (ELF) structure, strings, function call graph, Operation Code (OpCode) that could be used to detect malicious code. Generally, to classify a binary file, we need to understand the programs' functionalities and behaviour patterns. OpCodes are static discriminating features that can represent behavioural patterns of a binary file [3]. In the assembly language, an OpCode is a single instruction

[1]https://github.com/H-S-A/Embedded_ELF_Malware

command to be executed by the processor. OpCode sequence is a high-level static malware detection feature extracted from the binary file by a disassembler [13], [14] that usually comprises the program's execution logic [15].

Scalable, automated malware detection systems must efficiently detect malware without human expert analysis, and machine learning techniques could best fit this need. Deep learning (DL) is a sub-field of machine learning that uses deep neural networks to provide powerful statistical models. Popular DL models, such as convolutional neural networks (CNN) and long short-term memory (LSTM), have become prevalent due to their ability to learn features through deep multi-layer architecture with almost no need for data feature engineering [16]. CNN can learn essential signals from the raw data directly, retain an internal representation of the data, and do not require domain expertise to select the best features. LSTM, on the other hand, tends to remember long sentences of input data; thus it is often used as the encoder to create a contextual summary for the right prediction. However, LSTM does have flaws in remembering very long and complex sequences. Moreover, general deep neural networks cannot highlight some of the input OpCodes' importance than others during predictions.

The attention mechanism is one of the most recent valuable DL inventions. Recent innovations in natural language processing (NLP), such as Transformers and Google's Bidirectional Encoder Representations from Transformers (BERT) architectures[2], are based on attention. Attention mechanism can generate weight values for different words in a sentence, giving higher weights to the core words. Since OpCode sequence can be interpreted as a sentence, with each OpCode acting as a word in the sentence, the attention-mechanism can be used with OpCode sequences. Therefore, the attention mechanism can be combined with the DL models to extract critical information describing the context and the semantics of the OpCode sequence. The use of an attention-based embedding and a custom attention-based bidirectional long-short term memory (BiLSTM) allows the model to benefit from understanding the whole data context and extracting useful signals to improve the prediction task.

The transformer model [17] comprises an encoder-decoder architecture model based on attention mechanisms to transmit an entire picture of the whole sequence to the decoder at once. BERT uses bidirectional transformer-based encoders and employs masked language modeling to generate word embedding. The generated word embedding includes the word's contextualized meaning, thus makes the learning process more effective. Embedding is a low dimensional description of a point in a higher dimensional vector space. Similarly, BERT word embedding is a compact vector representation that translates a word to a lower-dimensional vector space to display the semantic significance and contextualized meaning of this word in a numeric form, thus allowing mathematical operations on text input.

In this paper, we propose DeepWare, a malware detection solution that detects malware targeting embedded systems by analyzing the executable OpCode sequence representations. We learn a malware representation using an embedder, an encoder, and a predictor. The embedder is composed of the state-of-the-art BERT embedding model [18] to capture relations within a single OpCode sequence and among the OpCode sequences in a trail. The encoder and the predictor construct a DL network named multi-head CNN-BiLSTM-LocAtt. The encoder incorporates three neural layers: a multi-head CNN layer that learns features and internal representations of the embedded data sequence directly without feature engineering. Followed by a BiLSTM layer, designed to preserve the ordinal position of an OpCode sequence in the trace, and a local-attention mechanism (LocAtt) [19] to compare interrelations among the different places of an OpCode sequence. Finally, the predictor consists of a dense layer and an activation layer that classifies the input as either malware or benign executable.

In this work, we consider three of the extensively used embedded processor architectures [20], [21]: (1) *ARM processor*: Largest current embedded IoT devices are based on top of an ARM processor. Hence, we have mainly chosen an ARM-based platform for all the evaluations; (2) *PowerPC processor*: it is used in many embedded solutions that can achieve one task in a single central processing unit (CPU) cycle. They are very popular among manufacturers of a system on a chip (SoC) and network devices, such as routers and switches. Recently, IoT malware has been discovered to target these network devices [22]; (3) *MIPS processor*: it has a modular architecture, and its instruction set architecture (ISA) has recently become open-source, thus increasingly being used in IoT devices [23].

Since a large number of embedded platforms use Linux as their operating system (OS) [24], we confine our work to detect Linux-based malware. To validate and evaluate our claims, we perform exhaustive experimentation focusing on the Linux environment with the three different types of processors on our collected datasets, which include 549 ARM, 877 PowerPC, and 749 MIPS embedded malware and benign samples. To the best of our knowledge, this is the first malware solution that uses DL networks and BERT architecture on OpCode sequences for detecting malware that targets embedded systems. The main contributions of this paper are as follows:

- We propose DeepWare, a real-time cross-architecture malware detection solution for embedded devices. DeepWare can run on either the network devices or edge devices to detect malware. We validate our method by collecting OpCodes datasets and running all the experiments on three of the most commonly used embedded microcontroller architectures: ARM, PowerPC, and MIPS.
- We propose to combine BERT sentence embedding and CNN together with BiLSTM DL techniques for embedded malware detection. We use the OpCode BERT embedding sequence to extract the fine-grained inherent connection and the semantic behaviors between embedded programs. These extracted features are then fed into a deep learning network based on CNN-BiLSTM, which considers the hidden malicious program behaviors to detect malware. We construct a malware identification network of multi-headed CNN and BiLSTM networks

[2]https://github.com/google-research/bert

for mining correlation and context information between different OpCodes, and a LocAtt mechanism to re-adjust the importance (weights) of the correlated features from OpCodes and their sequences to improve the deep learning ability to detect malware. We name the DL model multi-head CNN-BiLSTM-LocAtt. The LocAtt is seamlessly compounded with CNN-BiLSTM since the LocAtt weights for feature re-adjustion are calculated from both features and the hidden states of the CNN-BiLSTM. To the best of our knowledge, this is the first time a LocAtt mechanism is used for embedded devices' OpCode sequence malware detection.
- Finally, we publish an extracted and normalized OpCode dataset of IoT malware and benign applications for the three most common microcontroller architectures and we open sourced our multi-head CNN-BiLSTM-LocAtt model as a subsequent contribution. We believe this dataset and the code can contribute significantly to the development of the emerging embedded systems malware detection research area.

The rest of the paper is organized as follows. Section II provides an overview of recent IoT malware research. We then present our DeepWare solution, an automated cross-architecture OpCode sequence-based malware detection for embedded devices in Section III. This section includes the details on the proposed system architecture, intuition behind choosing each component of the model, dataset creation, feature extraction, and modeling details. Section IV reports the experimental studies and discusses the results of the proposed model. The results compare the proposed model and the state-of-the-art IoT malware detection techniques, and baseline machine and deep learning classifiers. Finally, Section V concludes the work with possible future improvements.

II. Related Work

A significant number of research works employing static, dynamic, hybrid, or memory analysis approaches have been conducted to analyze and detect malware or classify malware families [6], [24]–[26]. Many studies review, and provide comparisons between static and dynamic strategies, stating the advantages and disadvantages of both techniques [27], [28]. Several studies choose to use the static analysis method to investigate the overall malware composition without executing it [3], [5], [29].

Detecting malware in embedded systems is an emerging research topic, especially after Mirai's attacks [30] or its malware families. Machine learning and deep learning methods have been used in several static and dynamic analysis and detection methodologies. The conclusion of the aforementioned surveys and literature find that OpCode-based features are beneficial for malware detection, and there is currently a limited number of studies in the literature that target embedded or IoT devices malware detection.

Recently, DL (deep learning) models have been adopted to detect malware based on OpCodes for ARM-based IoT applications [1], [16], [20], [25]. The authors of [16] used OpCodes feature and Recurrent Neural Networks (RNN) to detect malware in IoT with a detection rate (DR) up to 98.18%. In [31], the authors converted the application binaries to gray-scale images and then used a CNN layer to detect and classify malware. They classified malware with an accuracy of 94.0%. Sharmeen et al. [32] examined static, dynamic, and hybrid analysis methods for industrial IoT malware. They showed that the permission list, API call list, and the system call list are significant features for distinguishing mobile malware. The authors of [20] counted the number of occurrences of ARM-based OpCodes to detect malware. They stated that the frequency of specific OpCodes in malware samples was higher than that of benign binaries. In [33], the authors applied fuzzy and fast fuzzy pattern tree methods on ARM-based OpCode sequences to detect and classify malware. They achieved almost perfect classification results.

Most of these studies used OpCode sequences and deep learning approaches, producing efficient and encouraging malware detection results. Nevertheless, there is an extent to reform the current malware detection models for different embedded systems architectures. Embedded devices are implemented with distinct CPU architectures and processors, such as MIPS, ARM, PowerPC, and SPARC. Most of the current works focus mainly on the ARM instruction set [1], [16], [20], [25]. There are differences between the instruction sets on different architectures. Therefore there is a necessity to establish an advanced cross-architecture static malware detection covering multiple architectures with limited processing time.

Moreover, many of the current OpCode sequence-based approaches extract n-grams from a sequence, which preserves some of the sequence information to certain extent but unfortunately loses considerable code semantic information [15].

III. DeepWare: A Malware Detection Solution

We propose an automated real-time malware detection solution, named DeepWare, for cross-architecture embedded systems. This section presents the proposed solution. We first introduce an overview of the DeepWare solution architecture. Afterward, we offer a detailed explanation for each of the building blocks of the proposed approach. Then we give details of our created datasets and all the steps taken to extract features. Finally, we describe the model layers and the data flow within each layer.

A. Overview DeepWare

The overall architecture of our DeepWare malware detection solution is shown in Fig. 1. Firstly, each sample's OpCodes sequence is extracted and applied to a fine-tuned and pre-trained BERT model to extract OpCode sequence embeddings. The extracted embedding preserves the contextual features of each OpCode in the sequence and transforms these features into a numeric form. The extracted features are then applied to a model consisting of a multi-head CNN-BiLSTM with local attention to extracting semantic information and correlations from the OpCode sequences. The model is trained and validated with OpCode sequence embeddings. After training the model, DeepWare can identify whether a binary file is a malware.

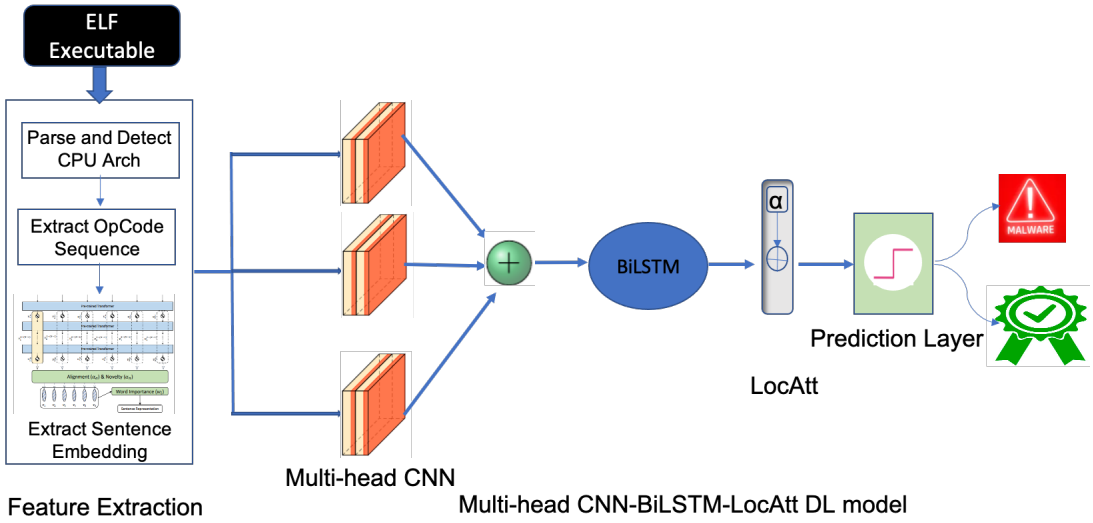


Fig. 1. The Architecture of Our DeepWare Malware Detection Solution.

B. Intuition Behind the Approach

1) BERT: Several adjoining OpCodes customarily establish a set of assembly operations with a particular context. Collecting these specific sets and analyzing them can be very beneficial in assessing samples and detecting malware. Many scholars used a varying slice n-grams to capture the associated relationship between subsequent OpCode sets. However, these approaches do not usually account for a meaningful set of instructions [34]. BERT uses deep bidirectional Transformers that employ a self-attention mechanism to trace relationships between all words in a sentence [35]. The sequence of OpCodes of an ELF binary file can be treated as a sentence, where each OpCode denotes a word (token). BERT uses the whole context and strives to predict the masked words' original value as per the OpCodes' context in the sequence. Accordingly, BERT pre-trained models can adequately create relationships between OpCodes in different contexts.

BERT looks at the sequence in both the left-to-right and the right-to-left paths to form bidirectional representations [34]. We can fine-tune pre-trained BERT representations for custom classification tasks. Subsequently, we use a fine-tuned BERT embedding's attention mechanism to capture an appropriate OpCodes sequence slice size and detect the selected sequences' global context.

The BERT embedding layer captures the critical OpCode information from the sample input and can efficiently learn the representation of relevant sequence patterns of the embedded systems malware. This layer maps the Opcode or OpCode sequences to an embedding vector, reflecting the semantic distance and the relationship between OpCodes. The embedding output generates different score values for different input values according to the global context [15].

2) Hybrid Multi-Head CNN-BiLSTM with Attention: Combining LSTM and CNN models has been used in various domains such as natural language processing (NLP) and anomaly detection with remarkable results [36], [37]. The core intuition behind stacking CNN with BiLSTM is to respect the local information in OpCode sequences and long-distance dependencies between OpCodes [36].

The multi-head convolution is a CNN that processes each input on an entirely self-governing convolution, known as convolutional heads. The multi-head CNN is responsible for extracting significant features from the input data. We consider a multi-head CNN layer with varying kernel sizes that moves vertically down for the convolution operation considering several OpCodes from the sequence together depending on the selected filter size. Otherwise, the extraction of features would be done on the whole input sequence, leaving each state's potential vital features uncaptured. The multi-head CNN layer considers words in close-range only. Hence, it does not consider the overall context in a particular text sequence [38]. On the other hand, LSTM is a type of RNNs that can remember information for an extended period. LSTMs can remember previous information using hidden states and connect it to the current task, allowing such networks to consider the long-term semantic of the sequence.

We choose to use the BiLSTM as it accesses both forward and backward information and captures the contextual information in both directions [39]. BiLSTM and its variants consider the sequence structure, but it does not usually give higher weight to more meaningful words. Since not all OpCodes have equal importance in the OpCode sequence, we use the local attention mechanism to create an aggregated representation of the important OpCodes in the horizontal sequence and extract the internal spatial relationship between OpCodes. Finally, we use a classification layer with two possible outputs: 1 in case the given sample is a malware or 0 for benign samples.

C. Dataset Creation and Feature Extraction

1) Dataset Creation: In our research study, we create three datasets, a dataset for each of the three most commonly used microcontroller architecture in embedded systems [20].

In the ARM dataset, we utilize the IoT malware and benign application dataset created by the authors of [16]. The

ARM dataset includes 247 IoT malware and 269 IoT benign Debian package files of 32-bit ARM-based IoT applications. Regarding the MIPS and the PowerPC datasets, we collect malware samples from the VirusShare[3] database. We collect the benign samples from Raspberry Pie II and architecture-specific compatible applications from the Linux repositories[4] and verify them by VirusTotal[5] to ensure benignity.

2) Feature Extraction: Algorithm 1 illustrates the feature extraction process from an ELF executable sample.

Files Parsing. The first step in parsing the collected binary files is to unpack and decompress the embedded application to obtain their assembly source code (ASM) files. We capture the binary CPU architecture from the ELF file metadata. Then, we employ Objdump [40] to disassemble the files. We use a predefined matching pattern for extracting OpCodes from the gathered malware and benign unpacked samples based on the architecture type as illustrated in Fig. 2. We read the .ASM file content and split each word. We then reserve the OpCode parts and discard the operand part. We compare each of the split words with the predefined OpCodes list related to the based architecture. We loop on all words in the input file, and if the word, in turn, matches any of the OpCodes in the list, the word is to be included in the final OpCodes sequence file. Using pattern-matching, we discard any meaningless OpCodes or wrong Opcodes that are not in the known pattern list.

Generating OpCode Sequences. It is worth mentioning that the way these OpCode sequences are obtained and prepared can dramatically undermines embedded malware detection methods' accuracy and complexity. Accordingly, we give extra consideration to the pre-processing steps to ensure all information is captured. Each sample in the dataset includes a sequence of assembly OpCodes. Each sequence of OpCodes is then transmitted to a vector $Op_s{}_1^n$, where n is the total number of OpCodes in a sample Op_s. Then we reformulate and split the collected long Opcode sequence vector of each sample into vectors of 512 consequent OpCodes Op_split_i. The remaining OpCodes within a sequence are considered in the next row of the array Op_split_{i+1}. This reformatting and splitting processes continue until the last OpCode Op_n within an OpCode sequence, creating a matrix $Op\_split_1^N$ of 512 sequences, where N is the number of raws within the created matrix. Hence all OpCodes within an application are taken into consideration as described in Eq. 1, 2:

$$Op_s = Op_1, Op_2, ...Op_n; \tag{1}$$

$$Op_s = Op\_split_1^N = \begin{bmatrix} Op_1 & Op_2 & Op_i & Op_{512} \\ Op_{513} & Op_{514} & Op_{split_l} & Op_{1024} \\ Op_j & Op_{j+1} & Op_{split_{j+l}} & Op_n \end{bmatrix} \tag{2}$$

[3]https://virusshare.com/
[4]https://rpmfind.net/linux/RPM/, https://pkgs.org/
[5]https://www.virustotal.com/

Algorithm 1: Feature Extraction for DeepWare

Input: ELF binary file
Result: Sentence embeddings of the OpCode sequence of the binary file, Emb(Op_split)$_1^N$

Function `Files Parsing(`$binaryfile$`)`**:**
  $F \longleftarrow binaryfile$;
  procedure UNPACK AND DECOMPRESS(F)
  return $ASMfile$;
  end procedure
  Function `disassemble(`$ASMfile$`)`**:**
    $Arch \longleftarrow detect(Binary - Architecture)$;
    Function `extract OpCode_sequence(`$ASMfile, Arch$`)`**:**
      while *not at the end of the file* **do**
        read current
        foreach $Op \in ASMfile$ **do**
          if $Op \in Arch$ **then**
            $Op_s \longleftarrow op$;
          end
        end
      end
      Op_s = Op_1, Op_2,...Op_n; where n is the total number of OpCodes in an executable.
    End Function
  End Function
  return Op_s;
End Function
Function `Generate OpCode_Sequences(`Op_s`)`**:**
  $i \longleftarrow 1$;
  foreach $Op_j \in Op_s$ **do**
    if $j < i * 512$ **then**
      Op_split$_i$ $\longleftarrow Op_j$;
    else
      $i \longleftarrow i + 1$;
      $Op\_split_i \longleftarrow Op_j$,
    end
  end
  Op_split = $Op\_split_1^N$, N is the number of (512 OpCode) rows within the array.
  Op_split_i is the i^{th} generated 512 length OpCode sequence vector.
  return Op_split;
End Function
Function `Generate Sentence Embedding(`Op_split`)`**:**
  foreach $Op\_split_i \in Op\_split$ **do**
    Generate tokenized output venctor, T(Op_split_i);
    Generate Sequence embedding, Emb(Op_split_i)
  end
  return $Emb(Op\_split)_1^N$;
End Function

Sentence Embedding Extraction. Each sample is applied to a BERT tokenizer to transform an OpCode sequence into a numerical vector while maintaining the context of the OpCode. We pad the output sequences vector to ensure all samples have

```
PowerPC_opcodes = ['add','addc','adde','addi','addic','addic.','addis','addme','addze','and','andc','andi.','andis.',
                   'b','bc','bcctr','bclr','cmp','cmpi','cmpl','cmpli','cntlzd','cntlzw','crand','crandc','creqv',
                   'crnand','crnor','cror','crorc','crxor','dcbf','dcbst','dcbt','dcbtst','dcbz','divd','divdu',
                   'divw','divwu','eciwx','ecowx','eieio','eqv','extsb','extsh','extsw','fabs','fadd','fadds',
                   'fcfid','fcmpo','fcmpu','fctid','fctidz','fctiw','fctiwz','fdiv','fdivs','fmadd','fmadds',
                   'fmr','fmsub','fmsubs','fmul','fmuls','fnabs','fneg','fnmadd','fnmadds','fnmsub','fnmsubs',
                   'fre','fres','frsp','frsqrte','frsqrtes','fsel','fsqrt','fsqrts','fsub','fsubs','hrfid','icbi',
                   'isync','lbz','lbzu','lbzux','lbzx','ld','ldarx','ldu','ldux','ldx','lfd','lfdu','lfdux','lfdx',
                   'lfs','lfsu','lfsux','lfsx','lha','lhau','lhaux','lhax','lhbrx','lhz','lhzu','lhzux','lhzx','lmw',
                   'lswi','lswx','lwa','lwarx','lwaux','lwax','lwbrx','lwz','lwzu','lwzux','lwzx','mcrf','mcrfs',
                   'mcrxr','mfcr','mfocrf','mffs','mfmsr','mfspr','mfsr','mfsrin','mftb','mtcrf','mtocrf','mtfsb0',
                   'mtfsb1','mtfsf','mtfsfi','mtmsr','mtmsrd','mtspr','mtsr','mtsrin','mulhd','mulhdu','mulhw','mulhwu',
                   'mulld','mulli','mullw','nand','neg','nor','or','orc','ori','oris','popcntb','rfid','rldcl','rldcr',
                   'rldic','rldicl','rldicr','rldimi','rlwimi','rlwinm','rlwnm','sc','slbia','slbie','slbmfee',
                   'slbmfev','slbmte','sld','slw','srad','sradi','sraw','srawi','srd','srw','stb','stbu','stbux',
                   'stbx','std','stdcx.','stdu','stdux','stdx','stfd','stfdu','stfdux','stfdx','stfiwx','stfs',
                   'stfsu','stfsux','stfsx','sth','sthbrx','sthu','sthux','sthx','stmw','stswi','stswx','stw',
                   'stwbrx','stwcx.','stwu','stwux','stwx','subf','subfc','subfe','subfic','subfme','subfze',
                   'sync','td','tdi','tlbia','tlbie','tlbsync','tw','twi','xor','xori','xoris']
```

Fig. 2. PowerPC-based Architecture OpCodes Pre-defined pattern List.

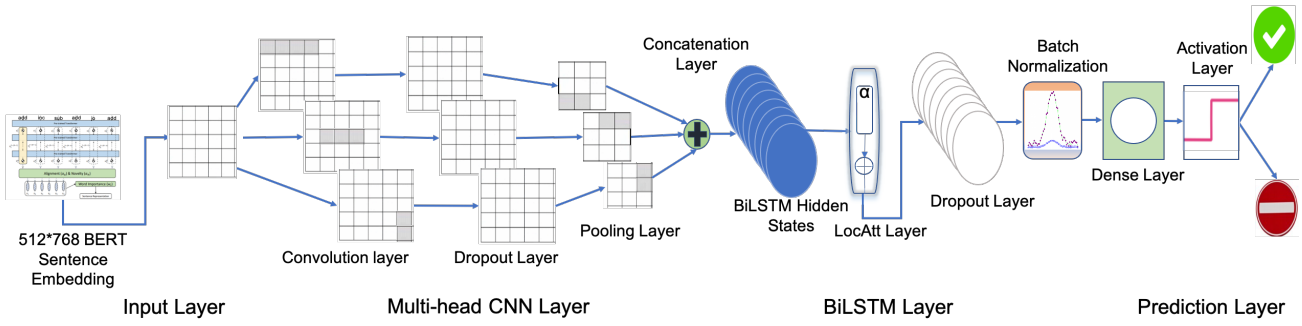


Fig. 3. Our Proposed Hybrid Multi-Head CNN-BiLSTM-LocAtt Model Design.

the same length. The tokenizer replaces each OpCode with its corresponding vector and creates a matrix $X \in R^{n*d}$ for the next operation, where n is the number of OpCodes with a sequence, and d is the dimension of output embedding.

Then we apply the tokenized data to a pre-trained BERT model embedder, specifically the BERT classification model from [17] to capture the relations and semantic within OpCode sequences. The BERT model has 12 hidden layers as explained in [18], and we collect and concatenate the tensor output of all the hidden layers. Each collected matrix output dimension is then reduced by squeezing the batch dimension, creating an output two-dimensional tensors vector of size *(512*768)* for Op_split_i. The constructed matrix is then concatenated with the generated tensor for the rest of the sequence vectors of a sample $Op\_split_1^N$. Finally, the generated embedding matrix $Emb(Op\_split)_1^N$ is applied to the neural network model for training and classification.

D. Modeling

We develop an attention-based multi-head CNN-BiLSTM hybrid model that combines BiLSTM and CNN powers with attention mechanism benefits. The model design is depicted in Fig 3.

The CNN model uses convolutional layers and maximum pooling layers to learn from the raw data directly to secure higher-level features without the need for domain expertise to manually engineer input features. In contrast, BiLSTM models capture long-term dependencies between OpCodes in sequences, maintaining the overall sequence semantics. We use a customized attention layer based on [19] after the BiLSTM layer to allow the classification layer to benefit from the overall context and to achieve better accuracy in the prediction. It is worth mentioning that we use a dropout layer with a 10% rate after each layer for regularization. To prevent overfitting, we use a batch normalization (BN) layer before the prediction layer. The BN layer is essential as it decreases the internal covariance shift [37] that regularizes batches and accelerate training [41].

1) Input Layer: A two-dimensional input vector (512*768) created by the BERT embedding layer is fed in this layer. Then a two-dimensional output is passed to the CNN layer for further feature extraction and parameter tuning.

2) Multi-head CNN Layer: We use multi-head CNN consisting of three parallel 1D CNNs heads to capture all the potential connections between adjacent OpCodes. We use filter sizes of 64 and kernel heads with varying sizes, precisely, a window size of 2, 3 and 4, to capture hidden associations

within different adjacent OpCodes [36]. The number of each layer's parameters on a multi-head CNN is computed using Eq. 3 and 4:

$$params = F_N * (K_S * P_P L + 1); \tag{3}$$

$$params = F_N * K_S * P_P L + bias \tag{4}$$

where F_N represents the number of filters, K_S denotes the kernel window size, P_{PL} represents the last output vector from the previous layer, where +1 in means we have one bias per filter accordingly $bias$ equals F_N.

The filter size is the number of times the input is processed, whereas the kernel size is the reading window size of the input sequence. Since we use a multi-head model, the model reads the input sequence using three different sized kernels, enabling the model to read and understand the sequence data at three different resolutions.

The CNN head is composed of three stacked one-dimensional layers, including the convolution, dropout and pooling layers. The convolution layer with rectified linear units (ReLU) activation [42] is the core component that detects and extracts meaningful features of the input sample. The dropout layer is used to slow down the learning process and prevent overfitting. Finally, the pooling layer is mainly used to reduce the training time and diminish overfitting by decreasing the number of parameters, consolidating them to only essential elements. We use max-pooling layers with $pool\_size = 2$. Max pooling skims a window over its input and carries the max value in the pooling window, thus, downsampling each feature map independently, reducing dimensionality, and keeping the depth intact. Each of the CNN head processes its input sequences window by window. A sequence of feature maps is then collected for each input sequence in a consecutive order. A feature map FM_1^w is captured for each window input data, where w denotes the number of windows (in our case $w = 3$). Then, the window's output is concatenated with each other to create a sequence of feature maps.

3) BiLSTM With Attention Layer: The features map of each window created by concatenating each convolutional head's output feeds the BiLSTM layer. In this way, the BiLSTM works on the extracted feature maps in chronological order, from FM_1 to FM_w. The BiLSTM processes each window to collect the relevant information corresponding to the current window in its internal memory.

The BiLSTM layer with ReLU activation function captures forward and backward contextual information and has a powerful ability to extract significant information, specifically in capturing long-distance OpCodes dependencies. However, the BiLSTM layer processes each word sequentially [43]; thus, they cannot extract the local context information. Therefore we use LocAtt layer after the BiLSTM layer to decide on the highly correlated features and diminishes the number of learnable weights required for the prediction.

The BiLSTM generates an output vector from the sequence of the hidden states $(h_1, h_2, ..., h_n)$ for each input sequence, where n is the number of embedded representations of OpCodes in a sequence. We capture each hidden state and sequences of the BiLSTM encoder by setting $return\_state = True$ and $return\_sequence = True$ for the Keras LSTM function. We then concatenate the backward and forward hidden states of the BiLSTM layer by applying them to a concatenation layer. The vector h_j represents the concatenation of the forward and backward hidden state of the BiLSTM encoder as represented in Eq. 5. The proposed customized attention mechanism is based on Bahdanau attention [19] that learns the network's weights and generates a context vector c_i for an output y_i using the weighted sum of the notes shown in Eq. 6 [19]. The weight α_{ij} are calculated using the softmax function in Eq. 7, where S_i is a BiLSTM hidden state for time i and e_{ij} is the attention score described by the function a that attempts to capture the association between input at j and output at i.

$$h_j = [^{\rightarrow}h_f^T, h_b^{T\leftarrow}]^T \tag{5}$$

$$c_i = \sum_{j=1}^{n} \alpha_{ij} * h_j \tag{6}$$

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{j=1}^{n} \exp(e_{ik})}; \qquad e_{ij} = \alpha(s_{i-1}, h_j) \tag{7}$$

We set the LocalAtt layer with a window size of 30. Then a dropout layer and a BN layer are applied to regularize the activations and accelerate the model training.

4) Prediction Layer: The prediction layer is a dense layer with one unit, used to capture the attention layer's final features. Then we follow it with an output layer with a Sigmoid activation function that predicts the final results.

IV. Experiments

To evaluate our approach, we use the real-world ELF embedded applications format. This section presents the dataset, evaluation metrics used in our experiments, experimental environment and the implementation details, and reports the experiments performed and their results.

A. Dataset and Evaluation Metrics

Dataset. As described in Section III-C1, we collect malware and benign samples from different architectures. We randomly divide our datasets into training sets (70%), validation sets (10%), and testing sets (20%). To generalize our experiments, we collect samples that include variants of the same sample from different architectures and different Linux and embedded Linux OS flavors. We use the training set to train the architectures, the validation set to tune them, and the test set to give us performance assurance. The test data set is selected large enough to provide statistically meaningful results and represent the whole dataset. Table I lists the number of training and test samples used in each CPU architecture experiment. We merge the number of validation and testing samples to represent samples used for testing in the table. The total split OpCode sequences represents the number of OpCode sequences created from the splitting operation as described in Section III-C2 and in Eq. 1 and Eq. 2.

TABLE I
STATISTICS OF THE MALWARE AND BENIGN SAMPLES DATASET.

ARM			
	Train	**Test**	**Total**
Malware	194	86	280
Benign	191	78	269
Total	385	164	**549**
Total Split OpCode Sequences	28,733	12,314	41,048
PowerPC			
	Train	**Test**	**Total**
Malware	496	193	689
Benign	121	67	188
Total	617	260	**877**
Total Split OpCode Sequences	20,060	8,600	28,660
MIPS			
	Train	**Test**	**Total**
Malware	273	127	400
Benign	251	98	349
Total	524	225	**749**
Total Split OpCode Sequences	34,965	14,785	49,750

Evaluation Metrics. This work is presented as a binary class classification problem to detect malware samples. Classification metrics such as receiver operating characteristics (ROC), False Positives (FP), False Negatives (FN), True Positives (TP) and True Negatives (TN) are used to show the accuracy and correctness of the proposed model. The precision metric confirms how many of the identified malware traces are correct, and the recall metric depicts how many of the malicious samples the model detects. Furthermore, the F1-score metric can unite both precision and recall metrics to show the average. The F1-measure relationship and the association between the correctly and incorrectly identified cases with F1-score are shown in Eq. (8):

$$\mathrm{F_1\text{-Score}} = \frac{2.precision.recall}{precision + recall} = \frac{2.TP}{2.TP + FP + FN} \quad (8)$$

B. Implementation Details

We use BERT to extract the sequence embedding and Tensorflow [44] to implement the model, and Keras [45] on top of Tensorflow to aid the implementation and experimentation. We test our model on a server with Core-i7 (3.6GHz) processor, 16 GB RAM, and a computer with Core-i5 (2.3GHz) processor, 8 GB RAM. Our experimental environment is as follows: Ubuntu 18.04.5, Python 3.6.7, CUDA 11.0, Transformers 3.0.2, Tensorflow 2.2.0, and Keras 2.4.3.

We build a functional CNN-BiLSTM model using the Keras deep learning library. The model expects a three-dimensional input with [batch_size, sequence_length, features_embedding_dimension]. We set the batch size to 20, and the epoch is 100 while setting early-stopping monitoring the validation loss with the patience of 15 epochs in case it converges before reaching the defined maximum number of epochs. The model's output is one vector containing the probability of a given window indicating malware or benign.

TABLE II
THE HYPER-PARAMETERS SPECIFICATION FOR ALL THE MODEL LAYERS.

Parameter	**Value**
Conv Heads	3
Conv. filter	64
Kernal window sizes	2,3,4
MaxPolling size	2
BiLSTM units	8
droupout	0.1
Attention Window Size	30
Output Dimension units	1
Learning rate	0.0001
epochs	100 (early stop is set)
batch size	20

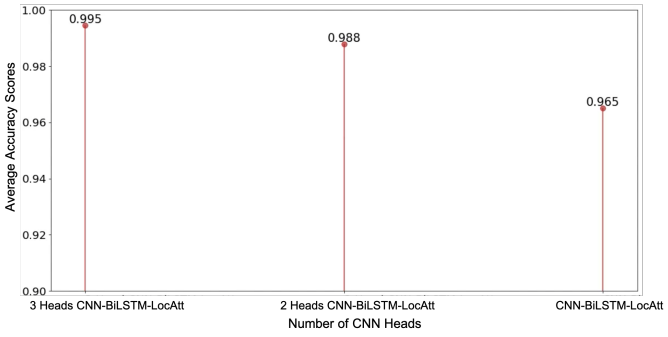
The training process is carried in a supervised manner, using back-propagation to propagate gradients from the final dense layer to the top convolutional layer. Since our concern is to detect malware, the architecture is designed for a binary classification task that uses binary cross-entropy [46] loss function. The selected model architecture can be implemented with a variety of CNN and RNN layer arrangements. The number of BiLSTM hidden layer nodes influences the complexity and performance results. The smaller the number of nodes, the limited learning ability of the network [47]. On the other hand, the larger the number of nodes, the more complex the network structure [47]. Regularly, there is no arrangement or parameter selection that best suits all scenarios. Therefore, a grid search determines which one performs best for the use case under consideration. We use the training set to train the architectures, the validation set to tune them, and the test set to give us performance assurance. We use Adam [48] as the optimizer with a learning rate of 0.0001. Table II depicts the model hyper-parameter specifications.

C. Results Discussion

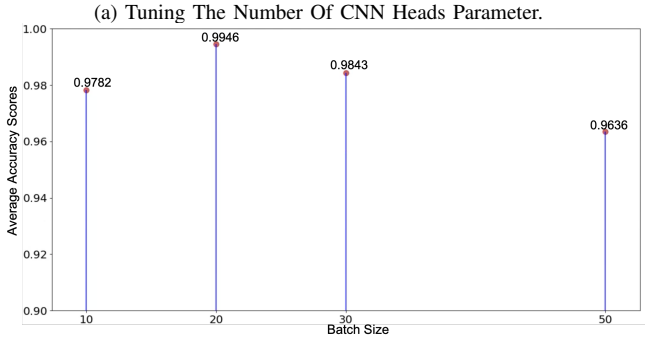
To examine and evaluate our proposed model's competency for embedded device malware detection, we conduct two different approaches to obtain comparable results. The primary experiment is on ARM-based dataset described in Section IV-C1, followed by the experiments in Section IV-C2 that include the results of our model when examined on the three different CPU architectures and comparison of the deep learning model with the baseline classification algorithms. The experiments are performed using 10 fold cross validation; while shuffling and splitting the datasets into training, validation, and test sets.

1) Preliminary ARM-based Results: In our preliminary experiments, we cover only the ARM-based embedded dataset collection to validate our proposed model's performance. We choose the ARM dataset to compare the results with the state-of-the-art methods [1], [16] because they are realized and tested on ARM datasets only. We train our model with 70% of the data and validate with the rest of the malware and benign samples. We use 10-fold cross-validation and achieve an average DR of 99.2%, FRR of 1.56%, and F1-score of 99.4%.

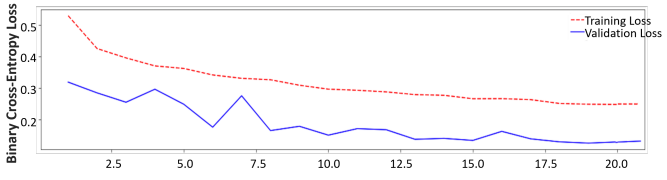
We conduct a set of experiments to select the optimal model parameters. Fig. 4 illustrates some of these experiments to choose the number of CNN heads, batch size, and the number of epochs. In particular, Fig. 4(a) presents a comparison between the number of heads in the multi-head CNN layer, showing that three heads give the best performance with a F1-score of 0.9946%. Fig. 4(b) illustrates the batch size selection experiment, where a batch size of 20 produces the best results. Fig. 4(c) shows that the lowest validation and training losses are achieved after 20 epochs. We set the ModelCheckpoint and the EarlyStopping parameters to monitor the validation loss to stop the training and save the best performing trained model.



(a) Tuning The Number Of CNN Heads Parameter.



(b) Tuning The Batch Size Parameters.



(c) Training Vs Validation Loss.

Fig. 4. ARM Dataset Different Parameter Setting Results. (a) Results for Different CNN Head Numbers, (b) Results for Different Batch Sizes, (c) Training and Validation Loss Through Out the Number of Epochs.

We compare our performance results with the recent malware detection studies in the literature that depends on OpCode sequence features for IoT-based malware detection using deep learning methods. The authors of [1] combined the information gain technique and the graph representation of selected OpCodes to create features. These features were then applied to Eigenspace and deep convolutional networks to classify samples. On the other hand, HaddadPajouh et al. [16] expressed that two layer LSTM network achieved the best accuracy to classify OpCodes samples. Table III displays the accuracy and F-measure scores comparing our model with recently presented OpCode based malware detection models.

TABLE III
COMPARISON BETWEEN THE PROPOSED MODEL AND THE STATE-OF-THE-ART OPCODE BASED MALWARE DETECTION APPROACHES.

Model	Accuracy	F1-Scores	Precision	Recall	Year
DeepWare	**99.39%**	**99.40%**	**99.40%**	**99.40%**	**2020**
Azmoodeh et al. [1]	99.68%	98.48%	98.59%	98.37%	2019
HaddadPajouh et al. [16]	94%	-	-	-	2018

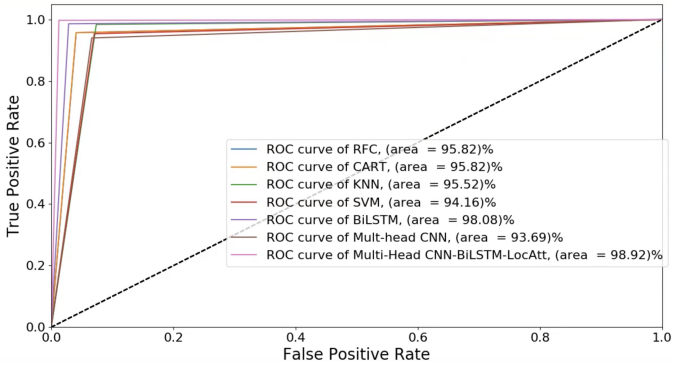


Fig. 5. ARM ROC Curve And AUC For Baseline Models In Comparison With Our Model.

By comparing our results with the ones from the state-of-the-art approaches, we slightly improve the detection accuracy for ARM-based malware detection. However, due to the lack of similar work in the literature that considers other CPU architectures, we can not present a comprehensive comparison. Accordingly, we validate our results with several baseline models, to be discussed in Section IV-C2.

2) Baseline Algorithms Comparison: In this section, we compare our results with baseline classifiers, namely Random Forest (RFC), Decision Tree (CART), k-Nearest Neighbor (KNN), Support Vector Machine (SVM), BiLSTM, and Multi-head CNN. We use $n\text{-}estimators$=100 for both RFC and CART. We use the default parameters for the selected baseline models. We run the processed OpCode sequence as detailed in Algorithm 1 through BERT and collect all the hidden states produced from all its layers. The tensor vector used for the DL model is with shape $512 * 768$, as illustrated in Fig 3. However, the baseline models require a one vector input. To get a single vector for our entire OpCode sentence, we choose to average all the hidden layers of each token, producing a single 768 length vector."We first compare our model and the baseline models using the ARM dataset. We then perform a comprehensive performance evaluation of our malware detection solution using all our CPU architecture datasets utilizing standards metrics, comparing our results to the baseline classification techniques. The ROC is a 2D space curve that shows the true-positive rate (TPR) and the false-positive rate (FPR) for each classifier. The area under the curve (AUC) is the area under the ROC curve and can evaluate a binary classifier's performance. Fig. 5 shows the comparison between the ROC and AUC for each classifier for the ARM

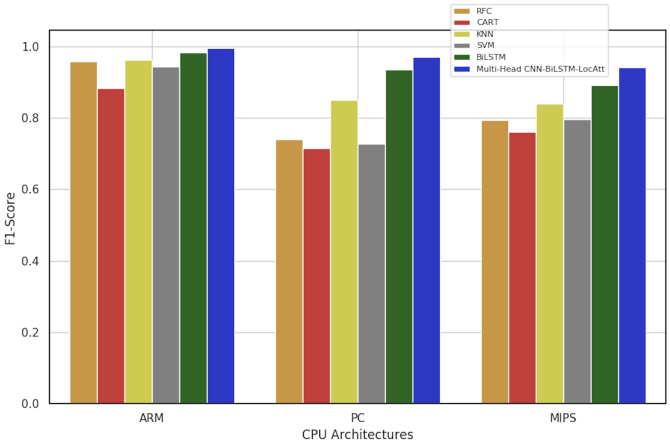


Fig. 6. A Comparison between Our Multi-Head CNN-BiLSTM-LocAtt Model and the Baseline Models. The Comparison is Conducted on the Three CPU Architecture Datasets.

dataset. The results indicate that our approach produces a lower FPR and a higher DR than all of the baseline classifiers.

Fig. 6 compares the classification F1-score of our model compared to the baseline classifiers for all of the CPU architectures in our dataset. Our proposed solution achieves F1-scores of 99.4%, 97.1%, and 94.2% for the ARM, PowerPC, and MIPS datasets, respectively. The results prove that our proposed approach is better than all the baseline classifiers with consistently high accuracy for all architectures.

Fig. 7 illustrates a comparison of the DR and the FRR achieved by our solution against the baseline techniques. Fig. 7(a) denotes the results for the ARM dataset experiments, and our approach achieves an almost perfect DR and a very low FRR of 1.26%. Fig. 7(b) shows the DR and FRR results for the PowerPC dataset. The results indicate that our model produces the highest DR of 98.99% and FRR of 8.7%. We observe high FRR results in this experiment, which is mainly due to the highly imbalanced data samples of the PowerPC dataset, as shown in Table I. Fig. 7(c) presents the experimental results on the MIPS dataset where our model achieves over 90% DR and less than 3% FRR.

To validate that our model can be used as a real-time detection solution, we run an experiment to compare the execution time taken to predict if an unknown binary file sample is a malware or benign using our model and the baseline models. Fig. 8 illustrates the experimental results showing that our deep-learning model is almost stable across all CPU architectures with an average of 62ms for prediction time. CART and RFC classify samples in the least amount of time, which is less than 10ms. Our model performs similarly to BiLSTM and slightly higher than multi-head CNN in the ARM experiments. However, it classifies PowerPC and MIPS samples slower than the both. Based on our experimental results, SVM and KNN are consistently the slowest classifying models. Moreover, OpCode extraction for an unknown binary file takes an average of 1.35sec. The sample embedding extraction time is avout 1.46sec. Accordingly the overall unknown binary file sample detection time using our model is of an average of 3.43 seconds.

TABLE IV
TRAINING EXECUTION TIME OF OUR MULTI-HEAD CNN-BILSTM-LOCATT MODEL IN COMPARISON TO THE BASELINE MODELS. THE TIME FORMAT IS HH:MM:SEC.MSEC.

Model	Training Elapsed Time
RFC	00:01:42.587812000
CART	00:01:20.742626000
KNN	00:01:02.437473000
SVM	11:12:19.223035000
BiLSTM	00:15:35.452159000
Multi-head CNN	00:11:45.130163
Our model	**00:15:30.978338000**

Regarding the training elapsed time, Table IV demonstrates the training time for our approach compared to the baseline models. The training of our model spends around 15 mins, with several seconds less than the BiLSTM model. SVM is the slowest model to train. On the other hand, KNN and CART are the fastest models to train, with fewer than 2 mins.

V. CONCLUSION AND FUTURE WORK

This paper proposes a novel real-time embedded systems' malware detection solution named DeepWare that uses CPU architectures, OpCode sequences, and deep learning (DL) mechanisms to detect malware. We extract BERT sentence embedding from the binary file OpCode sequences to maintain the sequence's semantic meaning and context. Then we apply the generated sentence embedding to a hybrid multi-head CNN-BiLSTM model with local attention to extracting the inherent and correlated features within a sequence. The multi-head CNN extracts the features of an OpCode sequence on an entirely independent basis. Afterwards, the BiLSTM extracts the long-term forward and backward relationship between OpCodes, passing the information to an attention layer (LocAtt) that determines the highly correlated features. Finally, a prediction layer is used to determine whether a sample is benign or malware. We implement and evaluate DeepWare on three different embedded CPU architectures: ARM, MIPS, and PowerPC. Our extensive evaluation results demonstrate the DeepWare's potential to efficiently detect embedded systems' malware, specifically embedded Linux malware variants targeting the three different CPU architectures. We prove through statistical tests that DeepWare using hybrid multi-head CNN-BiLSTM with attention architecture outperforms the traditional classification models. Moreover, we compare the results of the ARM dataset experiments with the other state-of-the-art OpCodes based malware detection models. The experimental results show an encouraging improvement in the detection accuracy for ARM-based malware.

For the future work, we plan to extend our approach to other embedded CPU architectures such as X86. We also plan to explore the idea of integrating more static analysis based features such as API call sequence in our model to enhance the results. Investigating other machine learning techniques and sentence embedding methods for embedded malware analysis is another main future research direction.

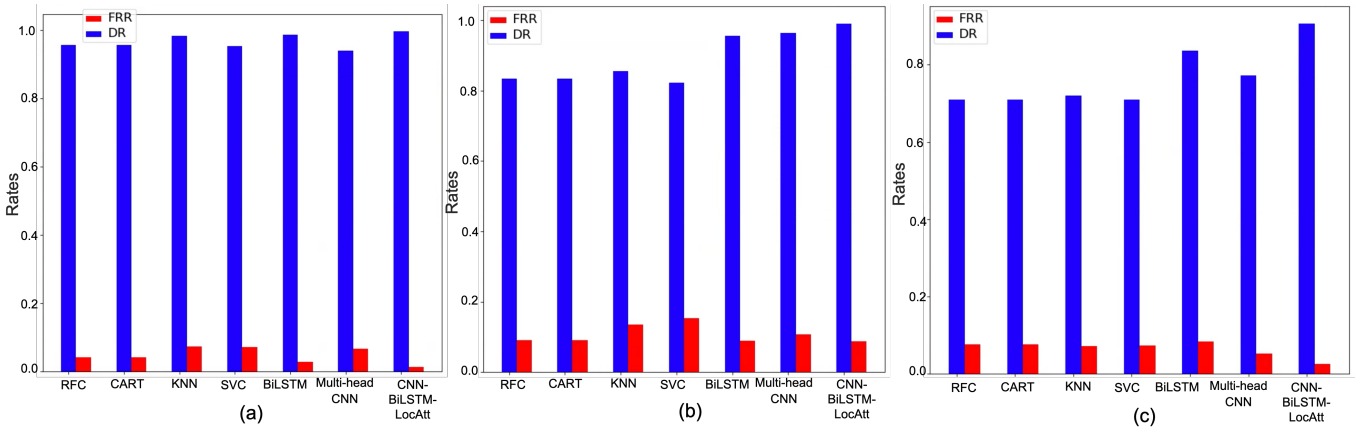


Fig. 7. Detection Rate (DR) and False Rejection Rate (FRR) For All The Baseline Models Against Our Model. (a) Results of ARM dataset, (b) Results of PowerPc dataset, (c) Results of MIPS dataset

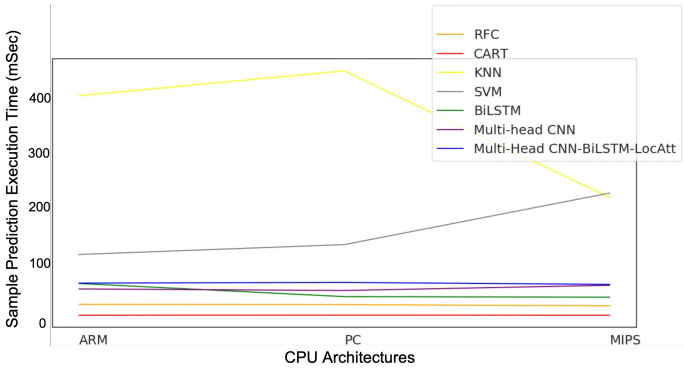


Fig. 8. Comparison of Prediction Time on the Three CPU Architecture Datasets: All Baseline Models vs Our Model.

REFERENCES

[1] A. Azmoodeh, A. Dehghantanha, and K. K. R. Choo, "Robust Malware Detection for Internet of (Battlefield) Things Devices Using Deep Eigenspace Learning," *IEEE Transactions on Sustainable Computing*, vol. 4, pp. 88–95, 2019.

[2] S. A. Hamad, Q. Z. Sheng, W. E. Zhang, and S. Nepal, "Realizing an Internet of Secure Things: A Survey on Issues and Enabling Technologies.," *IEEE Communications Surveys & Tutorials - COMST*, vol. 22, pp. 1372–1391, 2020.

[3] J. Zhang, Z. Qin, K. Zhang, H. Yin, and J. Zou, "Dalvik Opcode Graph Based Android Malware Variants Detection Using Global Topology Features," *IEEE Access*, vol. 6, pp. 51964–51974, 2018.

[4] K. Vasileios, K. Barmpatsalou, G. Kambourakis, and S. Chen, "A Survey on Mobile Malware Detection Techniques," *Transactions on Information and Systems - IEICE*, vol. E103-D(2), pp. 204–211, 2020.

[5] Z. Ren, H. Wu, Q. Ning, I. Hussain, and B. Chen, "End-to-end malware detection for android IoT devices using deep learning," *Ad Hoc Networks*, vol. 101, p. 102098, 2020.

[6] T. N. Phu, K. H. Dang, D. N. Quoc, N. T. Dai, and N. N. Binh, "A Novel Framework to Classify Malware in MIPS Architecture-Based IoT Devices," *Security and Communication Networks*, pp. 1–13, 2019.

[7] J. Jeon, J. H. Park, and Y. S. Jeong, "Dynamic Analysis for IoT Malware Detection with Convolution Neural Network Model," *IEEE Access*, vol. 8, pp. 96899–96911, 2020.

[8] W. Niu, R. Cao, X. Zhang, K. Ding, K. Zhang, and T. Li, "OpCode-Level Function Call Graph Based Android Malware Classification Using Deep Learning," *Sensors*, vol. 20, pp. 1–21, 2020.

[9] D. K. Hoang, D. Tho Nguyen, and D. L. Vu, "IoT Malware Classification Based on System Calls," *International Conference on Computing and Communication Technologies - RIVF*, pp. 1–6, 2020.

[10] A. Walbrecht, "BASHLITE malware turning millions of Linux Based IoT Devices into DDoS botnet," *Full Circle, The Independant Magazine For The Ubuntu Linux Community*, 2016.

[11] J. Biggs, "Hackers release source code for a powerful DDoS app, Mirai TechCrunch," 2016.

[12] H. Griffioen and C. Doerr, "Examining Mirai's Battle over the Internet of Things," *ACM Conference on Computer and Communications Security - CCS*, pp. 743–755, 2020.

[13] H. Rays, "IDA Home," 2020.

[14] Radare, "Radare2," 2020.

[15] X. M. Chen, S. Guo, H. Li, Z. Pan, J. Qiu, and Y. D. a. Feiqiong, "How to Make Attention Mechanisms More Practical in Malware Classification," *IEEE Access*, vol. 7, pp. 155270–155280, 2019.

[16] H. HaddadPajouh, A. Dehghantanha, R. Khayami, and K. K. R. Choo, "A deep Recurrent Neural Network based approach for Internet of Things malware threat hunting," *Future Generation Computer Systems*, vol. 85, pp. 88–96, 2018.

[17] Huggingface, "Transformers — transformers 4.0.0 documentation," 2020.

[18] J. Devlin, M.-W. Chang, K. Lee, K. T. Google, and A. I. Language, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *North American Chapter of the Association for Computational Linguistics: Human Language Technologies -NAACL-HLT*, vol. 1, pp. 4171–4186, 2019.

[19] D. Bahdanau, K. H. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *International Conference on Learning Representations - ICLR*, pp. 1–15, 2015.

[20] H. Darabian, A. Dehghantanha, S. Hashemi, S. Homayoun, and K. R. Choo, "An opcode-based technique for polymorphic Internet of Things malware detection," *Concurrency and Computation: Practice and Experience*, vol. 32, 2020.

[21] N. Shabab, "Looking for sophisticated malware in IoT devices — Securelist by Kaspersky," tech. rep., 2020.

[22] M. Kumar, "Linux worm targeting Routers, Set-top boxes and Security Cameras with PHP-CGI Vulnerability," 2013.

[23] W. Computing, "Wave Computing Launches the MIPS Open Initiative - WAVE Computing," 2018.

[24] S. P. Kadiyala, M. Alam, Y. Shrivastava, S. Patranabis, M. F. B. Abbas, A. K. Biswas, D. Mukhopadhyay, and T. Srikanthan, "LAMBDA: Lightweight Assessment of Malware for emBeddeD Architectures," *ACM Transactions on Embedded Computing Systems - TECS*, vol. 19, 2020.

[25] W. Peters, A. Dehghantanha, R. M. Parizi, and G. Srivastava, "A Comparison of State-of-the-Art Machine Learning Models for OpCode-Based IoT Malware Detection," in *Handbook of Big Data Privacy*, pp. 109–120, 2020.

[26] N. McLaughlin, A. Doupé, G. Joon Ahn, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, and Z. Zhao, "Deep Android Malware Detection," in *ACM Conference on Data and Application Security and Privacy - CODASPY*, pp. 301–308, 2017.

[27] Z. Bazrafshan, H. Hashemi, S. M. H. Fard, and A. Hamzeh, "A survey on heuristic malware detection techniques," *Conference on Information and Knowledge Technology*, pp. 113–120, 2013.
[28] Q. D. Ngo, H. T. Nguyen, V. H. Le, and D. H. Nguyen, "A survey of IoT malware and detection methods based on static features," *ICT Express*, vol. 6, pp. 280–286, 2020.
[29] H. Takase, R. Kobayashi, M. Kato, and R. Ohmura, "A prototype implementation and evaluation of the malware detection mechanism for IoT devices using the processor information," *International Journal of Information Security*, vol. 19, pp. 71–81, 2020.
[30] Josh Fruhlinger, "Mirai botnet explained," 2018.
[31] J. Su, V. Danilo Vasconcellos, S. Prasad, S. Daniele, Y. Feng, and K. Sakurai, "Lightweight Classification of IoT Malware Based on Image Recognition," *International Computer Software and Applications Conference - COMPSAC*, vol. 2, pp. 664–669, 2018.
[32] S. Sharmeen, S. Huda, J. H. Abawajy, W. N. Ismail, and M. M. Hassan, "Malware Threats and Detection for Industrial Mobile-IoT Networks," *IEEE Access*, vol. 6, pp. 15941–15957, 2018.
[33] E. M. Dovom, A. Azmoodeh, A. Dehghantanha, D. E. Newton, R. M. Parizi, and H. Karimipour, "Fuzzy pattern tree for edge malware detection and categorization in IoT," *Journal of Systems Architecture*, vol. 97, pp. 1–7, 2019.
[34] R. Oak, M. Du, D. Yan, H. Takawale, and I. Amit, "Malware detection on highly imbalanced data through sequence modeling," *ACM workshop on Artificial Intelligence and Security - AISec*, pp. 37–48, 2019.
[35] A. Vaswani, G. Brain, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Kaiser, and I. Polosukhin, "Attention Is All You Need," *Advances in Neural Information Processing Systems - NIPS*, vol. 30, pp. 5998–6008, 2017.
[36] B. Jang, M. Kim, G. Harerimana, S.-u. Kang, and J. W. Kim, "Bi-LSTM Model to Increase Accuracy in Text Classification: Combining Word2vec CNN and Attention Mechanism," *Applied Sciences*, vol. 10, p. 5841, 2020.
[37] M. Canizo, I. Triguero, A. Conde, and E. Onieva, "Multi-head CNN–RNN for multi-time series anomaly detection: An industrial case study," *Neurocomputing*, vol. 363, pp. 246–260, 2019.
[38] R. Agarwal, "NLP LEARNING SERIES (PART 3): Attention, CNN and what not for Text Classification," 2019.
[39] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, and Y. Tao, "A Survey on Deep Learning: Algorithms, Techniques, and Applications," *ACM Computing Surveys - CSUR*, vol. 51, pp. 1–36, 2018.
[40] Free Software Foundation, "objdump(1): info from object files - Linux man page," 2009.
[41] K. Zhang, W. Zuo, Y. Chen, D. Meng, and L. Zhang, "Beyond a Gaussian denoiser: Residual learning of deep CNN for image denoising," *IEEE Transactions on Image Processing*, vol. 26, pp. 3142–3155, 2017.
[42] H. Ide and T. Kurita, "Improvement of learning for CNN with ReLU activation by sparse regularization," *The International Joint Conference on Neural Networks - IJCNN*, pp. 2684–2691, 2017.
[43] M. Alazab, "Profiling and classifying the behavior of malicious codes," *Journal of Systems and Software*, vol. 100, pp. 91–102, 2015.
[44] Google, "TensorFlow," 2020.
[45] Keras, "Home - Keras Documentation," 2019.
[46] P. Golik, P. Doetsch, and H. Ney, "Cross-entropy vs. squared error training: a theoretical and experimental comparison," *undefined*, 2013.
[47] E. Batbaatar, M. Li, and K. H. Ryu, "Semantic-Emotion Neural Network for Emotion Recognition from Text," *IEEE Access*, vol. 7, pp. 111866–111878, 2019.
[48] Z. Zhang, "Improved Adam Optimizer for Deep Neural Networks," *IEEE/ACM International Symposium on Quality of Service - IWQoS*, pp. 1–2, 2018.

Salma Abdalla Hamad is currently an Information security manager at a financial institute. She received her PhD in 2021 at the School of Computing, Macquarie University. She worked under the supervision of Prof. Quan Z. (Michael) Sheng and Dr. Wei Emma Zhang. Salma obtained her Bachelor and master's degrees in Electronics and Communication Engineering from Arab Academy for Science, Technology, and Maritime Transport, Egypt with distinctions in 2005 and 2009, respectively. Salma also possesses 15+ years of working experience in both governmental sector and financial sector, where she executed several projects pertinent to Information Security. Her research interests are concentrated in the domain of Information Security, more specifically, for the Internet of Things, Smart Cities, and Smart Homes. Her research interests are concentrated in the domain of Information Security, more specifically, for the Internet of Things, Smart Cities, and Smart Homes.

Dai Hoang Tran is currently an AI engineer at Oracle. He received his PhD in the School of Computing, Macquarie University, MSc degree in Computing Engineering from Kyung Hee University, Korea, and his BS degree in Information Technology from Fontys University of Applied Science in Eindhoven, Netherlands, all in Computer Science, in 2011, 2015, 2021, respectively. He also works in the industry for more than 5 years as a software developer. His research interest focuses on applying analytic techniques of machine learning, data-mining and optimization to cutting-edge applications such as Big Data and the Internet of Things.

Quan Z. Sheng is a full Professor and Head of Department of Computing at Macquarie University, Sydney, Australia. Before moving to Macquarie, Michael spent 10 years at School of Computer Science, the University of Adelaide (UoA). Michael holds a PhD degree in computer science from the University of New South Wales (UNSW) and did his post-doc as a research scientist at CSIRO ICT Centre. His research interests include Internet of Things, service oriented computing, distributed computing, Internet computing, and pervasive computing. He has more than 400 publications, including ACM Computing Surveys, ACM TOIT, ACM TOMM, ACM TKDD, VLDB Journal, Computer (Oxford), IEEE TPDS, TKDE, DAPD, IEEE TSC, WWWJ, IEEE Internet Computing, Communications of the ACM, VLDB, ICDE, ICDM, CIKM, EDBT, WWW, ICSE. From 1999 to 2001, he also worked at UNSW as a visiting research fellow. Prof Michael Sheng is the recipient of several prestigious awards including AMiner Most Influential Scholar in IoT Award (2007-2017), ARC Future Fellowship in 2014, Chris Wallace Award for Outstanding Research Contribution in 2012, and Microsoft Fellowship in 2003. He is ranked by Microsoft Academic as one of the Most Impactful Authors in Services Computing (ranked Top 5 All Time) and in the Web of Things (ranked Top 20 All Time).

Wei Emma Zhang is currently a lecturer (Assistant Professor) in School of Computer Science, the University of Adelaide. Her research interests include the Internet of Things, text mining, data mining and knowledge discovery, and pervasive computing. Dr Wei Emma Zhang received PhD degree in computer science from the University of Adelaide in 2017 and did her post-doc at Department of Computing, Macquarie University, Sydney, Australia. She has authored and coauthored more than 70 papers. She has also served on various conference committees and international journals in different roles such as track chair, proceedings chair, PC member and external reviewer.