Intelligent Automation & Soft Computing
DOI:10.32604/iasc.2022.021038
Article

Tech Science Press

Massive IoT Malware Classification Method Using Binary Lifting

Hae-Seon Jeong[1] and Jin Kwak[2,*]

[1]ISAA Lab., Department of AI Convergence Network, Ajou University, Suwon, 16499, Korea
[2]Department of AI Convergence Network, Department of Cyber Security, Ajou University, Suwon, 16499, Korea
*Corresponding Author: Jin Kwak. Email: security@ajou.ac.kr
Received: 20 June 2021; Accepted: 23 July 2021

Abstract: Owing to the development of next-generation network and data processing technologies, massive Internet of Things (IoT) devices are becoming hyperconnected. As a result, Linux malware is being created to attack such hyperconnected networks by exploiting security threats in IoT devices. To determine the potential threats of such Linux malware and respond effectively, malware classification through an analysis of the executed code is required; however, a limitation exists in that each heterogeneous architecture must be analyzed separately. However, the binary codes of a heterogeneous architecture can be translated to a high-level intermediate representation (IR) of the same format using binary lifting and malicious behavior information can be identified because the functions and parameters of the assembly code are stored in the IR. Consequently, this study suggests a Linux malware classification method applicable to various architectures by converting Linux assembly codes into an IR using binary lifting and then learning the IR Sequence which reflects malicious behavior pattern using deep learning model for sequence learning.

Keywords: Assembly code; binary lifting; Linux malware; malware classification

1 Introduction

The development of next-generation network technology has enabled greater communication speeds and significantly reduced delays. As a result, a hyperconnected network environment called massive Internet of Things (IoT) has been developed, in which IoT devices used in a wide range of fields are interconnected and share information among each other [1].

However, a large number of IoT devices connected to a network have various vulnerabilities and can be subjected to security threats such as firmware forgery, unauthorized access by malicious applications, and information leakage through man-in-the-middle attacks. In addition, attackers can attack massive networks as well as massive numbers of IoT devices, causing enormous damage.

Linux malware, which can carry out highly potent malicious actions against such massive numbers of IoT devices, is very common nowadays and is spread through various types such as viruses, worms, trojan horses, and spyware. To minimize the damage from and effectively respond to Linux malware, it is important to determine various attack types through automatic malware classification. However, because IoT devices are composed of more than ten architectures, as shown in Tab. 1, it is difficult to apply a classification method using executed code analysis in a conventional Windows or mobile environment [2,3]. Hence, a separate analysis process is applied according to the instruction set architecture (ISA).

Table 1: Ratio of massive IoT malware attacks based on architecture [2]

Architecture	Number of samples	Ratio (%)
X86-64	3018	28.61
MIPS I	2120	20.10
PowerPC	1569	14.87
Motorola 68000	1216	11.53
Sparc	1170	11.09
Intel 80386	720	6.83
ARM 32 bit	555	5.26
Hitachi SH	130	1.23
ARM 64 bit	47	0.45
Others	3	0.03

At this time, if binary lifting is conducted, low-level assembly codes can be translated to a high-level intermediate representation (IR), which is a code of the same format regardless of the architecture, and information related to malicious behaviors can be identified because the functions and parameters used in the assembly codes are stored within the IR.

Therefore, in Section 2, we analyze the Linux file structure, automated Linux malware classification method, binary lifting technique, and B2R2 which is a concise and efficient binary lifting tool. Next, in Section 3, a Linux malware classification method using the IR is proposed. In Section 4, the results of the experiment conducted using the proposed classification method are analyzed. Finally, Section 5 provides concluding remarks on this research.

2 Related Works

2.1 Linux File

The executable file created by the compiler used in the Linux operating system (OS) is an executable and linkable format (ELF) file, which is a standard binary file commonly generated in various architectures and is composed of a sequence of bytes. When the ELF file is disassembled, as shown in Fig. 1, an assembly code that defines a low-level operation applied by the CPU is generated [4].

The details regarding the ELF file and assembly code are described in the following sections.

2.1.1 ELF File

The ELF file is composed of an ELF header, program header table, section, and section header table, as shown in Fig. 1. The ELF header stores information about the ELF file, such as the file type and architecture. The program header table stores information regarding the sections of the executable file. The section stores information regarding the object file, and the section header table stores the positions of all sections of the object file.

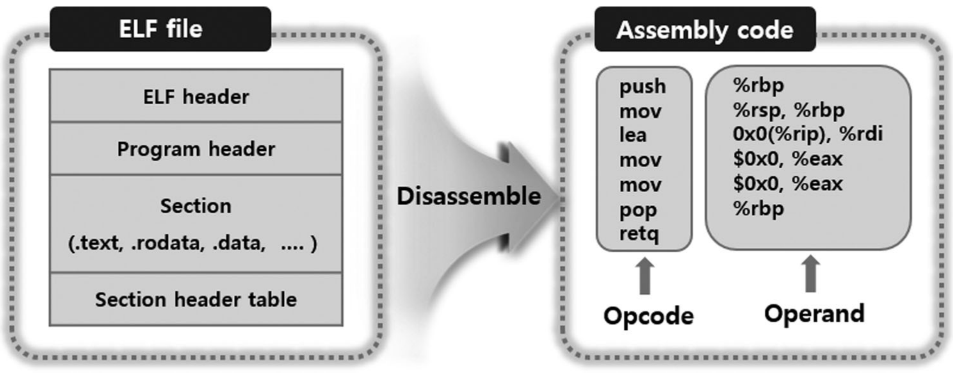


Figure 1: Structure of the ELF file and assembly code

Thus, the structures of the ELF and object files can be identified through the information stored in the ELF file. Various studies are currently being conducted currently to analyze the structural information of Linux malware using such files.

2.1.2 Assembly Code

An assembly code of the ELF file is composed of the opcode, which is the command of a specific architecture, and operand, which is a parameter used in operation. Thus, the assembly code reflects the program's execution flow.

Malware is a series of malicious behaviors, and many studies have been conducted to statistically analyze malware by extracting the opcode sequence in assembly codes to identify attack behaviors [5,6].

However, different analyses are required for each architecture when analyzing Linux files because they use the ISA, which defines the specifications of a unique opcode set for each architecture.

2.2 Linux Malware Types

Linux malware can attack mobile, IoT, and embedded devices, harming people and industries or stealing data from such devices. Linux malware can be classified into various types according to its basic functions and executed code [7,8].

In this study, malware is classified into seven types: backdoors, trojans, exploits, viruses, rootkits, worms, and hacktools. The detailed functions of each type are as follows:

- Backdoor: Backdoor-type Linux malware damages a system by bypassing authentication. After doing so, a backdoor program is installed to prevent users from recognizing the approach of an attack that may occur later. In 2019, Mirai backdoors were detected to be capable of infecting wide variety of devices including x64, x86, ARM and many other architectures [9].
- Trojan: Trojan-type Linux malware is designed to be similar to a normal program. However, once installed, it can lead to unauthorized access to a device, user behavior monitoring, data stealing, file transformation, and conversion of a user device into a botnet. Recently, trojans which perform distributed denial of services (DDoS) attacks are targeting devices which are running on systems supporting the ELF format [10].
- Exploit: Exploit-type Linux malware is a program that contains data or executable code that abuses any vulnerabilities by operating within a malicious program. It is used to hack a user device without a separate user operation in a net-worm. For example, Haiduc exploit toolkit was used for brute forcing the secure shell (SSH) for credentials using the given wordlist and scanning compromised devices by attackers [11].

- Virus: Virus-type Linux malware operates by activating infected software and infecting various devices connected to the same network through duplication. For example, Linux/Rst-B is a virus that attempts to download a page from a specific IP address if it is executed as root [12].
- Rootkit: Rootkit-type Linux malware is designed to bypass the recognition by device users and detection by security software and control a device through remote control. Linux malware, reported in 2014, had a loadable kernel module (LKM) rootkit component [13].
- Worm: Worm-type Linux malware duplicates itself, attacking devices through OS vulnerabilities, and spreads quickly. However, unlike a virus, it attacks and infects other devices, even if not executed by the user. For example, Worm.Linux.MALXMR.PUWELX is a worm which aims to permanently disable the system it infects arrives on a system as a file dropped by other malware or as a file downloaded and executes then deletes itself afterward [14].
- Hacktool: Hacktool-type Linux malware generates an authenticated user in a system and deletes the system logs related to malicious behaviors. Furthermore, they are used to collect and analyze network packets, and attackers can use a hacktool program to attack a device. For example, HackTool:Linux/BF.E reported in 2018 is a hacktool which arrives on a system as a file dropped by other malware or as a file downloaded but it does not have any propagation or backdoor routine [15].

2.3 Automated Linux Malware Analysis Method

Owing to the heterogeneous architecture of Linux executable files, building a dynamic analysis environment for them suffers from limitations related to resources. Hence, techniques to automatically analyze Linux malware by statically extracting information for analysis and learning such information through machine learning or a deep learning model are being studied. For static analysis, information other than the executable code, such as binary data and structure information of ELF files and text strings, is used. In addition, the opcode sequence related to the executed code and the control-flow graph (CFG) generated from it is also used. The existing studies on these analysis techniques are briefly introduced below.

Analysis methods based on information other than an executable code include deep learning after imaging the binary data of an ELF file or extracting and learning features from the ELF file. Kim et al. (2020) [16] converted an ELF file into images by generating a pixel of an 8-bit grayscale image using the binary data of the ELF file. The obtained images were then learned using a convolutional neural network (CNN)–based deep learning model to classify Linux malware. Hwang et al. (2019) [17] generated 200 feature vectors from the structural features of the ELF file, including the ELF header size, the number of program headers, and size of the section header, which were then learned using CNN-based deep learning models to classify the Linux malware. Wan et al. (2020) [6] generated feature vectors using the byte of the e_entry field entry point in the ELF header, which stores the packing information used during obfuscation and learned using a machine learning model for Linux malware classification. However, when the binary data and structural information of the ELF file are analyzed, the accuracy of the information analysis of malicious behaviors can be low because information on malicious behaviors directly conducted by malware is not analyzed.

The analysis method based on information related to executable code classified Linux malware by learning the opcode sequence extracted from the disassembled ELF file or the features in the CFG generated from the sequence. HaddadPajouh et al. (2018) [18] extracted features from an opcode sequence and learned using a long short-term memory (LSTM) deep learning model to detect IoT malware. However, this analysis method has a limitation in that it only analyzes the IoT malware of the ARM architecture and cannot detect heterogeneous Linux malware. Alasmary et al. (2019) [5] detected IoT malware by extracting 23 features, including the number of nodes and edges of the CFG, as well as the density and shortest path, and learned them using machine learning models. However, the accuracy of this approach can be low if it analyzes the Linux malware of various heterogeneous architectures because it analyzes such malware without considering the unique ISA of the architectures. With binary lifting, however, assembly codes of heterogeneous architectures can be translated into same format IR which has an efficient structure for evaluation. Therefore, in this paper, we propose heterogeneous Linux malware classification method by learning malicious behavior reflected in IR sequence using binary lifting which has not beenused for linux malware classification before.

2.4 Deep Learning Model for Sequence Learning

2.4.1 RNN

The recurrent neural network (RNN) model is an artificial neural network that solves problems by determining the size of the combination weights of neurons through the learning of a network formed by the weighted combination of artificial neurons (nodes). An RNN enables the learning of time-dependent relationships between the input data by enabling a combination of past information stored in the hidden layer with the current input values. However, this can give rise to a vanishing gradient problem for long input sequences. To solve this problem, the LSTM model has been proposed, the details of which are presented in the following section [19–21].

2.4.2 LSTM

The LSTM model is designed to solve the vanishing gradient problem of a Recurrent Neural Network (RNN), which is a deep learning model specialized for learning continuous sequence data with a temporal meaning. It can effectively learn long input sequences by learning the dependence between the input data [19–21]. The LSTM model is composed of three gates, namely, forget, input, and output, and learns the input sequences through a memory unit called a cell, which memorizes the information at each time point. The information to be memorized and the information to be forgotten is determined when the input information passes through each gate. The LSTM structure is in Fig. 2 and the terms used in the LSTM model are listed in Tab. 2. The detailed calculation process is presented below.

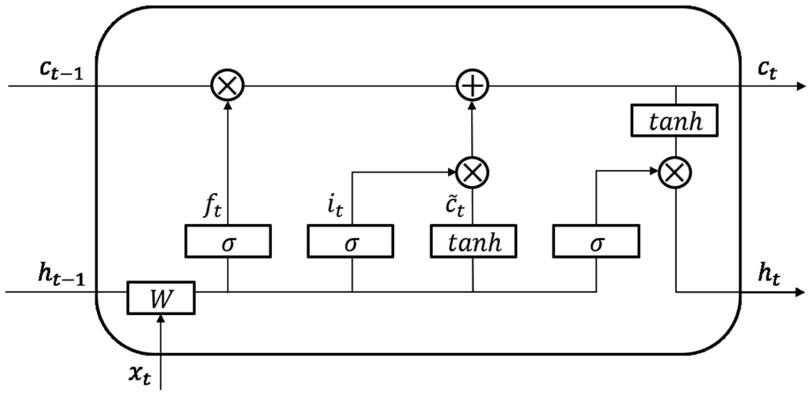


Figure 2: LSTM structure

Eq. (1) determines the degree to which the past information is memorized. Here, f_t produces a value between 0 and 1 as a resulting value by conducting a sigmoid operation for input variable x_t and the previous hidden variable h_{t-1}. The resulting value of 0 indicates that all previous information is forgotten, whereas 1 indicates that all previous information is memorized.

Table 2: Notations for LSTM structure

Notation	Description
t	Time in sequence
σ	Sigmoid function
x_t	Input state of time t
f_t	Forget state of time t
h_t	Hidden state of time t
c_t	Cell state of time t
o_t	Output state of time t
W	Weight sum
b	Bias sum
$tanh$	Hyperbolic tangent
$\otimes$	Element-wise multiplication
$\oplus$	Element-wise concatenation

$$f_t = \sigma\left(W_f \cdot [h_{t-1},\ x_t] + b_f\right) \tag{1}$$

Eqs. (2) and (3) are used to select new information by determining the information to be memorized. In Eq. (2), i_t has a value between 0 and 1 that determines which information needs to be memorized. In addition, $\tilde{c}_t$ in Eq. (3) is a candidate value that can be reflected in cell c_t and the result of the hyperbolic tangent calculation and thus has a value between −1 and 1. When the values of Eqs. (2) and (3) are determined, cell c_t is calculated using Eq. (4).

$$i_t = \sigma\left(W_i \cdot [h_{t-1},\ x_t] + b_i\right) \tag{2}$$

$$\tilde{c}_t = tanh\left(W_c \cdot [h_{t-1},\ x_t] + b_c\right) \tag{3}$$

$$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t \tag{4}$$

Eq. (5) determines the output o_t of the cell. Here, h_t is finally caculated by multiplying $tanh\ (C_t)$, which has a value between −1 and 1, with o_t through Eq. (6).

$$o_t = \sigma\left(W_o \cdot [h_{t-1},\ x_t] + b_o\right) \tag{5}$$

$$h_t = o_t * tanh\ (c_t) \tag{6}$$

2.4.3 LSTM Model for Malware Classification

Malware classification is influenced not only by the frequency of the machine language codes executed during the process but also by their patterns and trends. Hence, the LSTM model is appropriate for malware classification because the long-term pattern of using machine language codes must be learned to determine whether such a machine language code execution pattern corresponds to a specific type of malware.

2.5 Binary Lifting

Binary lifting is a technique for translating low-level assembly codes to high-level IR. Binary lifting can be conducted to standardize assembly codes composed of different opcode sets as IRs of the same format.

Consequently, in this study, the concise and efficient binary lifting tool B2R2 was analyzed and used to propose a heterogeneous Linux malware classification method [22].

2.5.1 Analysis on B2R2

B2R2 supports seven architectures, which occupy a high share among Linux malware. Furthermore, B2R2 efficiently conducts binary lifting through multi-core parallelism, using multiple CPUs by using the function-type programming language F#, which does not depend on the external environment [22].

2.5.2 Analysis on LowUIR

LowUIR, the IR of B2R2, stores information such as the variables used for efficient data analysis and the applied functions. The components of the LowUIR structure include METADATA, ENDIAN, and EXPRESSION, which specify the format, and UNOP, BINOP, RELOP, and CASTOP, which indicate the operational functions. STATEMENT is the highest unit that defines the functions of LowUIR using these components. The LowUIR generated by binary lifting has one or more STATEMENTs [22].

The usage formats of the components and elements listed in Tab. 3 are shown in Fig. 3. As LowUIR is designed to embed information about metadata and operations, assembly code can be quickly evaluated by analyzing the LowUIR.

Table 3: Components and elements of LowUIR

Component	Elements
METADATA	ExprInfo, ConsInfo
ENDIAN	BEndian, LEndian
UNOP	NEG, NOT
BINOP	ADD, SUB, MUL, DIV, SDIV, MOD, SMOD, SHL, SHR, SAR, AND, OR, XOR, CONCAT
RELOP	EQ, NEQ, GT, GE, SGT, SGE, LT, LE, SLT, SLE
CASTOP	ZeroExt, SignExt
EXPRESSION	Num, Var, PCVar, TempVar, Name, UnOp, BinOp, RelOp, Load, ITE, Cast, Extract, Undefined
STATEMENT	ISMark, IEMark, LMark, Put, Store, Jmp, CJmp, InterJmp, InterCJmp, SideEffect

For example, LowUIR, generated as a result of binary lifting for the ‘xor %rsp, %rbp’ assembly code of the X86-64 architecture, is shown in Fig. 4. This starts with ‘IsMark’ and ends with the ‘IEMark’ STATEMENT, and the EXPRESSIONs used in the ‘Put’ STATEMENT are ‘Var’ and ‘BinOp’ in order. Here, BINOP ‘XOR’ used in the ‘BinOp’ EXPRESSION means that the corresponding instruction performs an XOR operation.

3 Proposed Classification Method

The heterogeneous massive IoT malware classification method proposed in this study labels the collected malware samples as multiple classes of a normal program which is called benignware or malware types, as shown in Fig. 5. For this, the dataset to be used during the training and testing phases is randomly selected, and the LSTM model learns to classify Linux malware types. This is composed of a training phase during which the LowUIR sequence for the Linux ELF file is learned and classified using the LSTM model, and a testing phase during which the classification result is predicted for an unlabeled Linux ELF file generated as a result of learning using the LSTM model. The details of the data labeling and training and testing phases are presented below.

METADATA	μ	::=	ExprInfo * ConsInfo
		|	ExprInfo
ENDIAN	ϵ	::=	BEndian | LEndian
UNOP	$\diamond_u$	::=	NEG | NOT
BINOP	$\diamond_b$	::=	ADD | SUB | MUL | DIV | SDIV | MOD | SMOD | SHL | SHR | SAR | AND | OR | XOR | CONCAT
RELOP	$\diamond_r$	::=	EQ | NEQ | GT | GE | SGT | SGE | LT | LE | SLT | SLE
CASTOP	$\diamond_c$	::=	ZeroExt | SignExt
EXPRESSION	$\boldsymbol{exp}$	::=	Num *value size*
		|	Var *name size*
		|	PCVar *name size*
		|	TempVar *name size*
		|	Name *name*
		|	UnOp $\diamond_u$ $\boldsymbol{exp}$ $\boldsymbol{\mu}$
		|	BinOp $\diamond_b$ $\boldsymbol{exp}$ $\boldsymbol{exp}$ $\boldsymbol{\mu}$
		|	RelOp $\diamond_r$ $\boldsymbol{exp}$ $\boldsymbol{exp}$ $\boldsymbol{\mu}$
		|	Load $\boldsymbol{\epsilon}$ *size* $\boldsymbol{exp}$ $\boldsymbol{\mu}$
		|	ITE $\boldsymbol{exp}$ $\boldsymbol{exp}$ $\boldsymbol{exp}$ $\boldsymbol{\mu}$
		|	Cast $\diamond_c$ *size* $\boldsymbol{exp}$ $\boldsymbol{\mu}$
		|	Extract $\boldsymbol{exp}$ *pos size*
		|	Undefined *size*
STATEMENT	$\boldsymbol{stmt}$	::=	ISMark *addr len*
		|	IEMark *addr*
		|	LMark *name*
		|	Put $\boldsymbol{exp}$ $\boldsymbol{exp}$
		|	Store $\boldsymbol{\epsilon}$ $\boldsymbol{exp}$ $\boldsymbol{exp}$
		|	Jmp $\boldsymbol{exp}$
		|	CJmp $\boldsymbol{exp}$ $\boldsymbol{exp}$ $\boldsymbol{exp}$
		|	InterJmp $\boldsymbol{exp}$ $\boldsymbol{exp}$
		|	InterCJmp $\boldsymbol{exp}$ $\boldsymbol{exp}$ $\boldsymbol{exp}$
		|	SideEffect *SideEffect*

Figure 3: Usage format of LowUIR

```
[|ISMark (0UL,3u);
  Put
    (Var (64,5,"RBP",IntelRegisterSet<20, 0, 0, 0>),
     Var (64,4,"RSP",IntelRegisterSet<10, 0, 0, 0>)); IEMark 3UL|]
```

Figure 4: LowUIR of 'mov %rsp, %rbp' assembly code

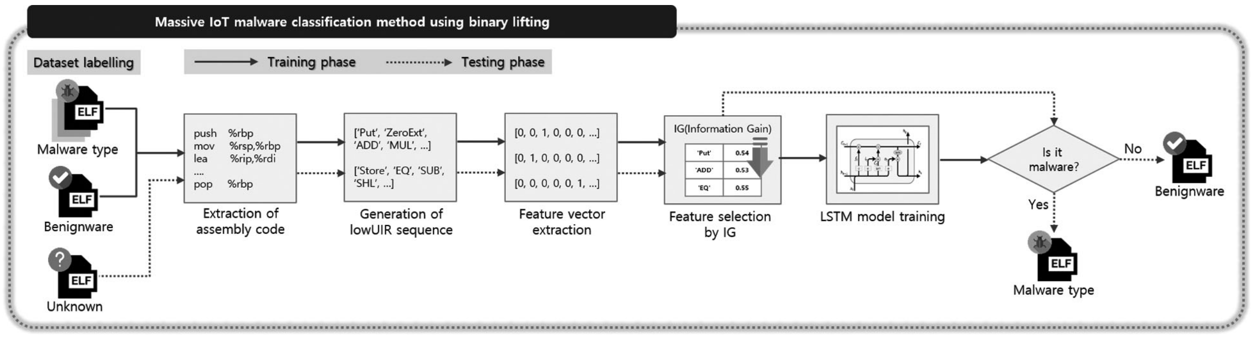


Figure 5: Proposed classification method

3.1 Dataset Labeling

The types of a dataset are labeled to train the LSTM model. Malware can be classified into multiple types. Because the malware detection results can differ by the anti-virus engine, we label the malware dataset as the type with the highest frequencies. To this end, the type that appears with the most high frequency on the Virustotal site, which shows malware detection results for each of the various anti-virus engines, is selected and labeled by using it.

For the labeled dataset, a value of 0 is assigned to benignwares, and a value within the range of 1 to N (where N is the number of malware types) is assigned to each type of malware for the indexing of each class.

3.2 Training Phase

During the training phase, features are generated from the LowUIR sequence created through binary lifting for a labeled ELF file. Next, features with high importance in terms of class identification are selected. The input values composed of the features are then learned using the LSTM model and classified into benignware or malware types. A detailed description of each step of the training phase is provided below [23].

3.2.1 Extraction of Assembly Code

The ELF file is disassembled to extract the assembly codes, which are the input data for binary lifting. The assembly codes are extracted using the disassembler tools.

3.2.2 Generation of LowUIR Sequence

The LowUIR sequence is extracted by applying binary lifting using B2R2 for the extracted assembly code sequence. In consideration of the time required to perform binary lifting, the LowUIR of the assembly code for each architecture is stored such that the LowUIR sequence can be extracted by matching it with the LowUIR of the stored assembly code.

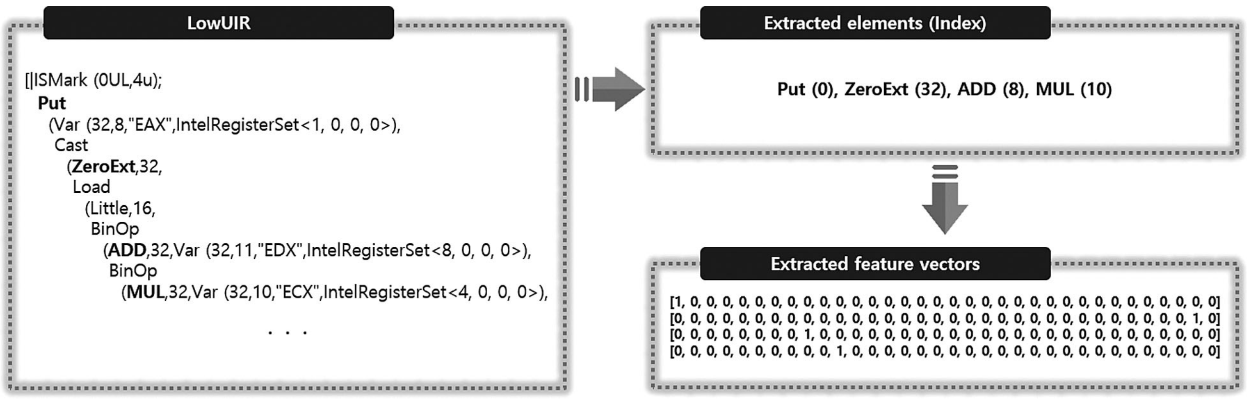
3.2.3 Feature Vector Extraction

The feature vectors are extracted from the elements listed in Tab. 4 for STATEMENT, which is used as the highest unit for the extracted LowUIR sequence, and UNOP, BINOP, RELOP, and CASTOP, which indicate different operations. As a result, documents composed of multiple words are generated, and encoding is conducted by assigning a unique index to each element, as shown in Tab. 4. To this end, feature vectors are extracted by applying one-hot encoding by assigning a value of 1 to the index if there is an element corresponding to each index and a value of zero if there is no such element.

Table 4: Components and elements of feature vector

Component	Elements (Index)
STATEMENT	Put (0), Store (1), Jmp (2), CJmp (3), InterJmp (4), InterCJmp (5)
UNOP	NEG (6), NOT (7)
BINOP	ADD (8), SUB (9), MUL (10), DIV (11), SDIV (12), MOD (13), SMOD (14), SHL (15), SHR (16), SAR (17), AND (18), OR (19), XOR (20), CONCAT (21)
RELOP	EQ (22), NEQ (23), GT (24), GE (25), SGT (26), SGE (27), LT (28), LE (29), SLT (30), SLE (31)
CASTOP	ZeroExt (32), SignExt (33)

An example of extracting the feature vectors according to Tab. 4 is shown in Fig. 6.



```
[|ISMark (0UL,4u);
  Put
    (Var (32,8,"EAX",IntelRegisterSet<1, 0, 0, 0>),
     Cast
       (ZeroExt,32,
        Load
          (Little,16,
           BinOp
             (ADD,32,Var (32,11,"EDX",IntelRegisterSet<8, 0, 0, 0>),
              BinOp
                (MUL,32,Var (32,10,"ECX",IntelRegisterSet<4, 0, 0, 0>),
                         . . .
```

Put (0), ZeroExt (32), ADD (8), MUL (10)

```
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Figure 6: Example of feature vectors extracted from LowUIR

3.2.4 Feature Selection using IG

To remove features that generate noise in the type classification and to reduce the size of the total feature set, the IG value for each feature, based on the notations defined in Tab. 5, is obtained using the information gain (IG) in Eq. (7). Those features with an IG value above the threshold are selected [24–28].

$$IG(f_m,\ C) = \sum_{i=0}^{j} -C_j \ln C_i \; - \; \sum_{q=\{0,1\}} \frac{|C_q|}{|C|} \sum_{i=0}^{j} -C_i \ln C_i \tag{7}$$

Table 5: Notations for IG

Notation	Description
m	Index of feature vector
f_m	m^{th} feature vector
C	Number of documents
j	Number of classes
C_i	Number of documents belonging to class i
C_q	Number of documents containing the feature vector f_m
C_j	Number of documents of class i containing the feature vector f_m
q	Presence or absence of a feature (0 if absent or 1 if present)

3.2.5 LSTM Model Training

The LSTM model proposed in this study is composed of an input layer, an embedding layer, a bidirectional LSTM (Bi-LSTM) layer, a dropout layer, and a dense layer, as shown in Fig. 7 below.

In the input layer, the feature vector generated as a result of one-hot encoding for benignware and malware is input and delivered to the embedding layer. Because the input feature vector is a sparse vector where the value of every index except for one is 0, the embedding layer generates a dense vector of the matrix type. In the Bi-LSTM layer, time-series data are processed and learned with high efficiency in both the forward and backward directions [29–31]. Here, LSTM repetitive learning is maintained by inserting the dropout layer before and after the generation of the Bi-LSTM layer, and the overfitting of the neural network is reduced [32]. Because multi-class classification is the final goal, the softmax activation function is used in the last dense layer [33].

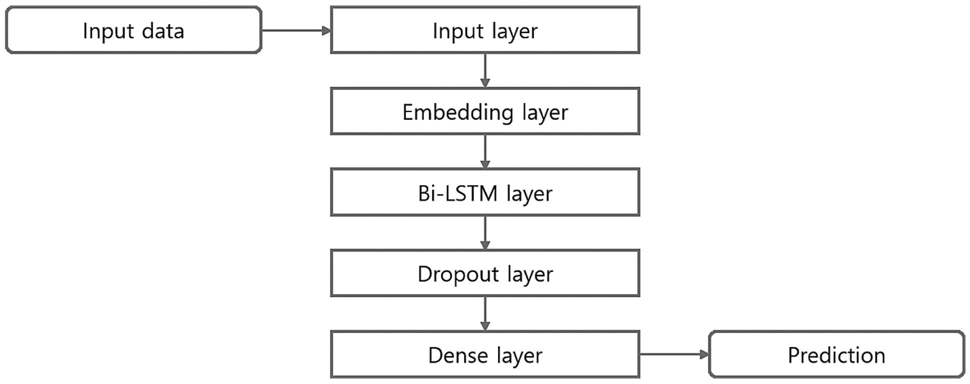


Figure 7: LSTM model structure

3.3 Testing Phase

During the testing phase, unlabeled ELF files are classified into benignware or malware types using the trained LSTM model. Each step of the proposed method is described in detail in the following section.

4 Experimental Analysis

4.1 Experiment Design

4.1.1 Dataset Collection

A dataset is created using the heterogeneous Linux malware sample provided for malware analysis from a public organization, and the class is labeled with the malware type that appears most frequently in the detection result of the Virustotal.

In the case of a benignware, the data are collected and labeled using open samples such as IoT device firmware.

The number of samples based on the malware type and the number of benignware samples for the Intel 80386, X86-64, MIPS, and ARM architectures are shown in Tab. 6.

Table 6: Number and ratio of samples

Malware type	Number of samples	Ratio (%)
Backdoor	3336	49.2
Trojan	1271	18.8
Exploit	587	8.7
Virus	578	8.5
Rootkit	258	3.8
Worm	129	1.9
Hack	95	1.4
Benignware	523	7.71
Total	6777	100

4.1.2 Experimental Setup

The Linux malware classification experiment conducted in this study is implemented using the programming language Python 3.8 in the Ubuntu 20.04 environment. The assembly codes are converted into a hexadecimal and input into the B2R2 0.3.0 tool. Malware is classified by learning the resulting IRs.

The details about the experimental setup are shown in Tab. 7.

Table 7: Experimental Setup for our proposed method

Category	Item
Hardware	Intel(R) Xeon(R) W-2123 CPU
	48GB Memory
	NVIDIA GeForce RTX 2080 GPU
Software	B2R2 0.3.0
	.NET SDK 5.0.101
	Ubuntu 20.04 LTS
	Python 3.8

4.1.3 Model Configuration

In this study, the LSTM model is implemented using the 'Keras' module, a neural network API that works with TensorFlow, an open-source machine learning framework. To this end, parameters that produce a high performance are selected by creating the LSTM model using the parameters within the search range listed in Tab. 8. The parameters selected in Tab. 8 were used to experiment the model proposed.

Table 8: Selected model parameters

Parameter	Search space	Selected parameter
Batch size	5–300	50
Number of epochs	10–20	10
Length of LowUIR sequence	2000, 4000, 6000, 8000	2000
Percentage of data to be used in testing	10–20	15
LSTM units	16, 32, 64, 128, 256	16
Embedding vector length	16, 32, 64, 128, 256	16
Dropout amount	0.1, 0.2, 0.3, 0.4	0.2

4.1.4 Evaluation Metrics

The classification performance of the proposed method is defined through accuracy in terms of true positive (TP), false positive (FP), true negative (TN), and false negative (FN); detailed definitions and explanations of these indices are as follows [28].

- TP: Number of malware classified as benignware
- FP: Number of benignware classified as malware types

- TN: Number of benignware classified as benignware
- FN: Number of malware classified as benignware or other malware types

Accuracy indicates the ratio of the input dataset that has been accurately classified, and is defined as Eq. (8):

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{8}$$

4.2 Analysis on Experimental Results

In this section, to analyze and verify the performance of the proposed method, the performances of the conventional heterogeneous Linux malware classification methods are measured together using information other than the executable code and compared with the performance of the proposed method. To this end, the accuracies of the classification method through the imaging of the ELF binary data [16] and the classification method using the structural information in the ELF [17] are measured and their performances analyzed.

The experimental results of Tab. 9 showed that the accuracies of all conventional classification methods were lower than 86%, whereas the accuracy of the proposed method was 94.2%.

Table 9: The experimental results

Classification method	Accuracy (%)
Our proposed method	94.2
The classification method based on ELF binary image [16]	85.4
The classification method based on ELF structural features [17]	82.8

The above results proved that heterogeneous Linux malware classification methods using the information other than the executable code have a lower accuracy compared with a classification method that analyzes information related to an executable code, which is similar to the proposed method, and that the proposed method can accurately classify various types of heterogeneous Linux malware.

5 Conclusion

The emergence of a massive IoT environment has increased the number of network vulnerabilities and attacks. Consequently, the frequency and types of heterogeneous Linux malware targeted at massive IoT devices are continuously increasing. Hence, a method for automatically classifying heterogeneous Linux malware is required. However, because each architecture has a unique ISA, methods used to automatically analyze the information related to the executable code of heterogeneous malware are lacking. Therefore, in this study, a Linux malware classification method is proposed that uses binary lifting to convert the Linux malware of heterogeneous architectures into the IRs of a common malware type that reflects the applied functions and learns them using an LSTM model.

The proposed method showed higher accuracy than the classification methods using the information other than the executed code. Furthermore, we demonstrated the improved efficiency and accuracy of the heterogeneous Linux malware classification method.

Therefore, the safety of a massive IoT environment can be improved by detecting massive IoT malware quickly and accurately using the heterogeneous Linux malware classification method proposed in this study.

In addition, the proposed method can be applied to malware family classification by subdividing the malware types. Furthermore, new and variant Linux malware can also be predicted using this method.

Funding Statement: This work was supported by a National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. 2021R1A2C2011391) and this work was supported by the BK21 FOUR program of the National Research Foundation of Korea funded by the Ministry of Education (NRF5199991514504).

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] I. Ahmad, S. Shahabuddin, T. Kumar, J. Okwuibe, A. Gurtov *et al.,* "Security for 5G and beyond," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 4, pp. 3682–3722, 2019.

[2] E. Cozzi, M. Graziano, Y. Fratantonio and D. Balzarotti, "Understanding Linux malware," in *2018 IEEE Sym. on Security and Privacy*, San Francisco, CA, USA, pp. 161–175, 2018.

[3] T. Ban, R. Isawa, K. Yoshioka and D. Inoue, "A cross-platform study on IoT malware," in *2018 Eleventh Int. Conf. on Mobile Computing and Ubiquitous Network*, Auckland, New Zealand, pp. 1–2, 2018.

[4] D. J. Jeon and D. G. Park, "Real-time Linux malware detection using machine learning," *Journal of Korean Institute of Information Technology*, vol. 17, no. 7, pp. 111–122, 2019.

[5] H. Alasmary, A. Khormali, A. Anwar, J. Park, J. Choi *et al.,* "Analyzing and detecting emerging internet of things malware: A graph-based approach," *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8977–8988, 2019.

[6] T. L. Wan, T. Ban, Y. T. Lee, S. M. Cheng, R. Isawa *et al.,* "IoT-malware detection based on byte sequences of executable files," in *Proc. 2020 15th Asia Joint Conf. on Information Security*, Taipei, Taiwan, pp. 143–150, 2020.

[7] S. Das, Y. Liu, W. Zhang and M. Chandramohan, "Semantics-based online malware detection: Towards efficient real-time protection against malware," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 2, pp. 289–302, 2016.

[8] Malwarebytes Labs, "2020 state of malware report," 2020. [Online]. Available: https://www.malwarebytes.com/resources/files/2020/02/2020_state-of-malware-report-1.pdf.

[9] S. Chatterjee, "Mirai-backdoor threat report," 2020. [Online]. Available: https://www.subexsecure.com/pdf/malware-reports/June-2020/Linux_Mirai-Backdoor.pdf.

[10] P. Kalnai and J. Horejs, "DDoS trojan: A malicious concept that conquered the ELF format," 2016. [Online]. Available: https://www.virusbulletin.com/uploads/pdf/conference/vb2015/KalnaiHorejsi-VB2015.pdf.

[11] Trend Micro Cyber Safety Solutions Team, "Perl-based shellbot looks to target organizations via c&c," 2018. [Online]. Available: https://documents.trendmicro.com/assets/Perl-Based_Shellbot_Looks_to_Target_Organizations_via_C&C_appendix.pdf.

[12] H. Martin, "The case for AV for Linux: Linux/Rst-B," 2008. [Online]. Available: https://www.virusbulletin.com/virusbulletin/2008/08/case-av-linux-linux-rst-b.

[13] Avast Threat Intelligence Team, "Linux DDoS trojan hiding itself with an embedded rootkit," 2015. [Online]. Available: https://blog.avast.com/2015/01/06/linux-ddos-trojan-hiding-itself-with-an-embedded-rootkit.

[14] M. Malubay, "Worm.Linux.malxmr.puwelx," 2021. [Online]. Available: https://www.virusbulletin.com/virusbulletin/2008/08/case-av-linux-linux-rst-b.

[15] K. Obuchi, "Hktl_sshbrute.ga-ELF," 2018. [Online]. Available: https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/HKTL_SSHBRUTE.GA-ELF.S.

[16] S. Kim, D. Kim, H. Lee and T. Lee, "A study on classification of CNN-based Linux malware using image processing techniques," *Journal of the Korea Academia-Industrial Cooperation Society*, vol. 21, no. 9, pp. 634–642, 2020.

[17] J. Hwang and T. Lee, "Study of static analysis and ensemble-based Linux malware classification," *Journal of the Korea Institute of Information Security & Cryptology*, vol. 29, no. 6, pp. 1327–1337, 2019.

[18] H. HaddadPajouh, A. Dehghantanha, R. Khayami and K. Choo, “A deep recurrent neural network based approach for internet of things malware threat hunting,” *Future Generation Computer Systems*, vol. 85, no. 4, pp. 88–96, 2018.

[19] F. A. Gers, J. Schmidhuber and F. Cummins, “Learning to forget: Continual prediction with LSTM,” in *1999 Ninth Int. Conf. On Artificial Neural Networks*, Edinburgh, UK, vol. 2, pp. 850–855, 1999.

[20] S. Hochreiter and J. Schmidhuber, “LSTM can solve hard long time lag problems,” in *Proc. Advances In Neural Information Processing Systems*, Cambridge, MA, USA, pp. 473–479, 1996.

[21] Q. Jing, H. Wang and L. Yang, “Study on fast-changing mixed-modulation recognition based on neural network algorithms,” *KSII Transactions on Internet and Information Systems*, vol. 14, no. 12, pp. 4664–4681, 2020.

[22] M. Jung, S. Kim, H. Han, J. Choi and S. Cha, “B2R2: Building an efficient front-end for binary analysis,” in *Proc. Workshop on Binary Analysis Research (BAR)*, San Diego, CA, USA, pp. 1–10, 2019.

[23] S. Basole, F. Di Troia and M. Stamp, “Multifamily malware models,” *Journal of Computer Virology and Hacking Techniques*, vol. 16, no. 1, pp. 79–92, 2020.

[24] G. Forman, “BNS feature scaling: An improved representation over TF-IDF for SVM text classification,” in *Proc. 17th ACM Conf. on Information and Knowledge Management*, New York, NY, USA, pp. 263–270, 2008.

[25] X. Ma, S. Guo, H. Li, Z. Pan, J. Qiu *et al.,* “How to make attention mechanisms more practical in malware classification,” *IEEE Access*, vol. 7, pp. 155270–155280, 2019.

[26] J. Kang, S. Jang, S. Li, Y. S. Jeong and Y. Sung, “Long short-term memory-based malware classification method for information security,” *Computers & Electrical Engineering*, vol. 77, no. 12, pp. 366–375, 2019.

[27] Y. Ding, W. Dai, S. Yan and Y. Zhang, “Control flow-based opcode behavior analysis for Malware detection,” *Computers & Security*, vol. 44, no. 2, pp. 65–74, 2014.

[28] A. Kapoor and S. Dhavale, “Control flow graph based multiclass malware detection using bi-normal separation,” *Defence Science Journal*, vol. 66, no. 2, pp. 138–145, 2016.

[29] Y. Chi and J. Li, “Video saliency detection using bi-directional LSTM,” *KSII Transactions on Internet and Information Systems*, vol. 14, no. 6, pp. 2444–2463, 2020.

[30] Y. Sung, S. Jang, Y. Jeong and J. H. Park, “Malware classification algorithm using advanced Word2vec-based Bi-LSTM for ground control stations,” *Computer Communications*, vol. 153, no. 12, pp. 342–348, 2020.

[31] M. Schuster and K. Paliwal, “Bidirectional recurrent neural networks,” *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[32] E. Andrade, J. Viterbo, C. Vasconcelos, J. Guerin and F. Bernardini, “A model based on LSTM neural networks to identify five different types of malware,” *Procedia Computer Science*, vol. 159, pp. 182–191, 2019.

[33] X. Cheng, H. Wang, J. Hua, M. Zhang, G. Xu *et al.,* “Static detection of control-flow-related vulnerabilities using graph embedding,” in *2019 24th Int. Conf. on Engineering of Complex Computer Systems (ICECCS)*, Guangzhou, China, pp. 41–50, 2019.