EvilELF: Evasion Attacks on Deep-Learning Malware Detection over ELF Files

Andrew Kosikowski
Rose-Hulman Institute of Technology, IN

Daniel Cho
Hamilton College, NY

Mabon Ninan, Anca Ralescu, Boyang Wang
University of Cincinnati, OH

***Abstract*—This paper investigates evasion attacks on end-to-end deep-learning malware detection over ELF (Executable and Linkable Format) binaries. We show that an attacker can deliberately modify bytes in a malware ELF binary such that a well-trained neural network is misled and predicts it as benign. We examine five methods that can modify ELF binaries without affecting functionalities and leverage them in evasion attacks. We explore two state-of-the-art end-to-end deep learning malware detectors, including MalConv and FireEyeNet, over a real-world dataset with 1,422 ELF binaries. Our experimental results show that evasion attacks with 3 out of the 5 methods are effective and can force the two CNNs to predict incorrectly. For instance, the most effective modification achieves up to 76.6% evasion rate on FireEyeNet and 8.4% evasion rate on MalConv. We also demonstrate that retraining CNNs with deliberately modified binaries can significantly mitigate evasion attacks.**

I. Introduction

End-to-end deep learning malware detection is a new approach of detecting malicious binaries [1], [2]. Specifically, all the bytes from a binary are formulated as a vector and utilized as an input to a malware detector, where the malware detector is a neural network. Compared to existing *static analysis* methods, end-to-end deep learning malware detection does not need to perform time-consuming feature engineering.

Despite the promising results, recent studies suggest that end-to-end deep learning malware detection is vulnerable under *evasion attacks* [3], [4], [5], [6], [7], [8], [9]. In an evasion attack, an adversary intentionally modifies certain bytes in a malicious binary such that the modified binary still carries the same functionalities but can force a well-trained neural network to predict incorrectly (more specifically, outputting benign rather than malware). Evasion attacks have been successfully demonstrated over PE (Portable Executable) binaries in Windows [3], [4], [5], [6], [7], [8], [9].

In this paper, we investigate evasion attacks on end-to-end deep learning malware detection over *ELF binaries*, which have not been well-investigated. Compared to PE binaries, ELF binaries used in Linux are more comprehensive in terms of structures, and therefore, more challenging to modify. Specifically, we examine *black-box* evasion attacks, in which an adversary does not have access to the details (weights or hyperparameters) of a neural network but can query it with various modified binaries and obtain associated predictions. Our findings are summarized below:

- We examine five modification methods, referred to as Header Alteration, Debug Alteration, Padding Alteration, End Appendix, and Dynamic Extension, which can modify bytes in ELF binaries without affecting functionalities.
- We explore two state-of-the-art end-to-end deep learning malware detectors, including MalConv [1] and FireEyeNet [2], over a real-world dataset containing 1,422 ELF binaries (711 benign and 711 malware). Experimental results show that the baseline detectors can achieve promising results in malware detection when there are no evasion attacks. For instance, MalConv can achieve a 95.5% F1 score and 99.6% AUC (Area Under the Curve).
- We demonstrate that evasion attacks using modified ELF binaries are effective. Specifically, evasion attacks with the most effective modification method – Padding Alteration – can achieve up to **76.6%** evasion rate on FireEyeNet and **8.4%** evasion rate on MalConv.
- We find that Padding Alteration, End Appendix and Dynamic Extension are all able to evade baseline detectors successfully. On the other hand, Header Alteration and Debug Alterations are not effective.
- We find MalConv is much more resilient than FireEyeNet, where an attack on it achieves a much lower evasion rate. For instance, evasion rate only reaches 1.6% on MalConv with an input size of 1 million bytes. We also show that retraining malware detectors with deliberately modified ELF binaries is an effective way to mitigate evasion attacks, especially over MalConv (e.g., mitigating evasion rate to 0.2% or less).

Reproducibility. Our source code and dataset can be found at https://github.com/UCdasec/EvilELF.

II. Related Work

White-box evasion attacks [3], [4], [5], [6], [7], [8], [9] have been proposed in the context of end-to-end malware detection over PE binaries. White-box attacks require an attack knowing the details of a neural network while black-box attacks do not.

Specifically, Kolosnjaji et al. [3] proposed an evasion attack against MalConv by padding optimized values at the end of each input. Demetrio et al. [4] designed a similar evasion attack against MalConv by modifying bytes in the DOS header. Kreuk et al. [5] developed a gradient-based evasion attack that perturbs bytes in either the slack space or the end-of-file space. This attack can achieve around 30% evasion rate against MalConv. Suciu et al. [6] further improved the evasion rate to 70% against MalConv based on the work in [5]. Sharif et al. [10]

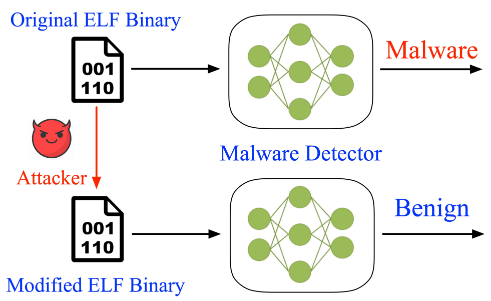


Fig. 1: System and threat model

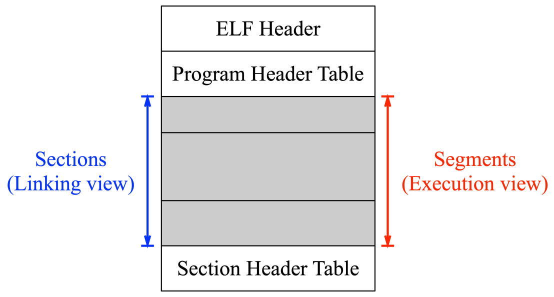


Fig. 2: The high-level structure of an ELF binary

proposed an attack to defeat neural-network based malware detectors by transforming the instructions, more specifically, binary diversification, without breaking functionalities. This method applies in-place randomization to replace opcodes inside a .text section with semantic equivalent opcodes or uses the *jump* function to move opcodes into a different section without altering original functions. Liu et. al. [11] leveraged different modifications over binaries to evade multiple neural networks simultaneously. A more comprehensive survey on evasion attacks over PE binaries can be found in [12]. *However, these attacks are all based on modifications over PE binaries in Windows and cannot be directly applied to ELF binaries in Linux.*

One recent study [13] proposed two methods to maliciously modify ELF binaries for evasion attacks on end-to-end deep learning malware detection. Their first method modifies zero bytes that are padded to the end of a binary when the size of a binary is less than the input size of a neural network. However, it only perturbs data in the input space and no real-world modified binaries are generated.

Their second method inserts a new section between two sections by modifying section offsets in the Section Header Table. However, offsets in the Program Header Table are not modified correspondingly. As a result, the modification may result in ELF binaries that are unable to execute. *Compared to [13], our work is able to produce deliberately modified binaries that can still run in the real world.*

III. Background

A. System and Threat Model

System Model. The system model includes a malware detector, which is a neural network. An input to a neural network is a vector of all the bytes from an ELF binary. The output is either 0 (benign) or 1 (malicious). All the binaries utilize the same input size, which is defined in advance. The input size is the number of bytes in a vector passed to a neural network. If the actual number of bytes in a binary is less than the input size, `0x00`s are padded at the end. If the actual number of bytes in a binary is greater than the input size, additional bytes beyond the input size are trimmed.

Threat Model. A ***black-box adversary*** can deliberately modify a malicious ELF binary, generate a modified malicious ELF binary, pass the modified ELF binary to the malware detector, and obtain the prediction result (either 0 or 1). The goal of this adversary is to evade the malware detector, such that the malware detector will predict the modified malicious ELF binary as benign. This is referred to as an *evasion attack*. Black-box indicates that the adversary does not know the details of the neural network but can submit modified ELF binaries and obtain associated prediction results.

Metric. We use accuracy, precision, recall, F1 score, and Area Under the Curve (AUC) to measure the performance of a malware detector. In addition, we use ***evasion rate*** to measure the effectiveness of an attack. Evasion rate is defined as the ratio between the number of modified binaries bypassing a neural network and the total number of modified binaries generated by an adversary.

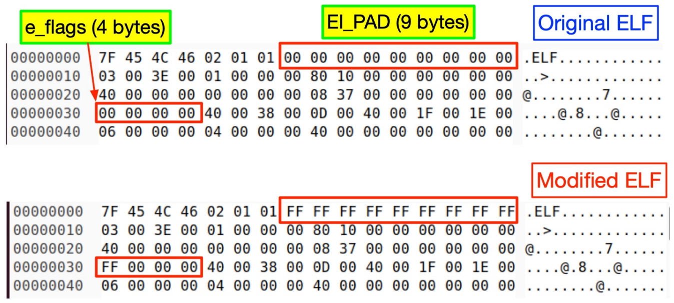
B. Structure of ELF Binaries

Executable and Linkable Format (ELF) is a standard executable file format typically used in Unix/Linux operating systems. An ELF binary normally consists of multiple components, including the ELF Header, the Program Header Table, segments/sections, and the Section Header Table [14]. the ELF Header is strictly defined at the beginning of an ELF binary while the locations of other components are arbitrary and are defined in the ELF Header. A high-level description of the ELF format is illustrated in Fig. 2.

ELF Header. An ELF Header consists of 52 or 64 bytes for 32-bit or 64-bit binaries respectively. The first few bytes in an ELF Header contains information regarding file classes, data encoding, object file types, architecture, and version information. In addition, an ELF Header also contains the offsets and sizes of the Program Header Table and Section Header Table, the number of sections in the ELF binary, etc.

Program Header Table. The Program Header Table describes segments contained in an ELF binary. It defines the number of segments contained in the ELF binary, the offset of each segment, etc. It also enables the operating system to execute the binary by informing where specific segments need to be loaded into memory.

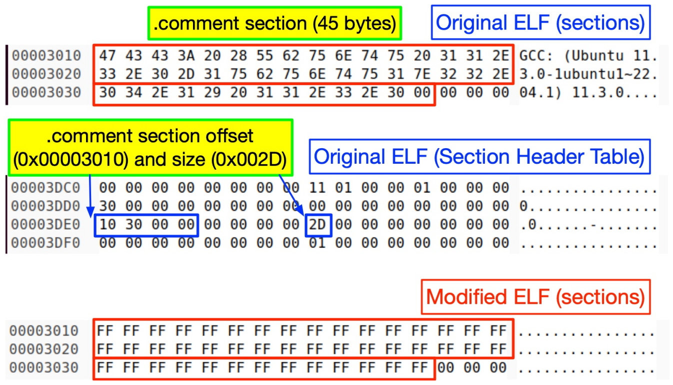
Segments/Sections. ELF offers two logical views/organizations over the same data within a binary: the execution view and the linking view. The execution view, which is organized mainly based on segments, informs the operating system how to load segments when executing an ELF binary. The linking



```
00000000  7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00  .ELF............
00000010  03 00 3E 00 01 00 00 00 80 10 00 00 00 00 00 00  ..>.............
00000020  40 00 00 00 00 00 00 00 08 37 00 00 00 00 00 00  @........7......
00000030  00 00 00 00 40 00 38 00 0D 00 40 00 1F 00 1E 00  ....@.8...@.....
00000040  06 00 00 00 04 00 00 00 40 00 00 00 00 00 00 00  ........@.......
```

```
00000000  7F 45 4C 46 02 01 01 FF FF FF FF FF FF FF FF FF  .ELF............
00000010  03 00 3E 00 01 00 00 00 80 10 00 00 00 00 00 00  ..>.............
00000020  40 00 00 00 00 00 00 00 08 37 00 00 00 00 00 00  @........7......
00000030  FF 00 00 00 40 00 38 00 0D 00 40 00 1F 00 1E 00  ....@.8...@.....
00000040  06 00 00 00 04 00 00 00 40 00 00 00 00 00 00 00  ........@.......
```

Fig. 3: An example of Header Alteration: all the 9 padding bytes and the first byte of `e_flags` are modified to `0xFF`.


.comment section (45 bytes)
Original ELF (sections)
.comment section offset (0x00003010) and size (0x002D)
Original ELF (Section Header Table)
Modified ELF (sections)

```
00003010  47 43 43 3A 20 28 55 62 75 6E 74 75 20 31 31 2E  GCC: (Ubuntu 11.
00003020  33 2E 30 2D 31 75 62 75 6E 74 75 31 7E 32 32 2E  3.0-1ubuntu1~22.
00003030  30 34 2E 31 29 20 31 31 2E 33 2E 30 00 00 00 00  04.1) 11.3.0....
```

```
00003DC0  00 00 00 00 00 00 00 00 11 01 00 00 01 00 00 00  ................
00003DD0  30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  0...............
00003DE0  10 30 00 00 00 00 00 00 2D 00 00 00 00 00 00 00  .0......-.......
00003DF0  00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00  ................
```

```
00003010  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
00003020  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
00003030  FF FF FF FF FF FF FF FF FF FF FF FF FF 00 00 00  ................
```

Fig. 4: An example of Debug Alteration: all the 45 bytes of `.comment` section are modified to `0xFF`.

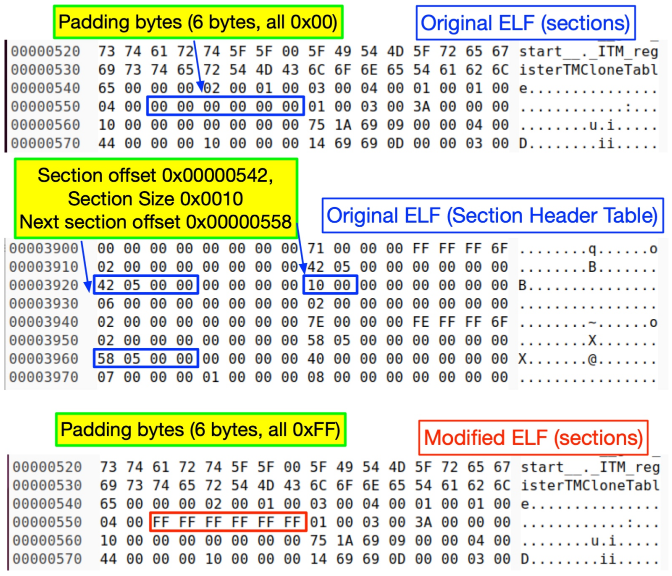
view, which is based on sections, offers information (e.g., metadata for debugging) at the link time.

Common sections include initialized data (.data), version control (.comment), dynamic linking information (.dynamic), symbolic debugging information (.debug), executable instructions (.text), a string table (.strtab), and a symbol table (.symtab). A segment contains one or multiple sections. Common segments include a loadable segment (PT_LOAD) and the dynamic linking segment (PT_DYNAMIC).

Section Header Table. The Section Header Table defines the size and offset of each section in an ELF binary and contains all information about the contents of a file. It is not loaded during the program execution, but it is necessary for linking and creating the original files.

IV. Modifications on ELF Binaries

Modifying an ELF binary without affecting its functionalities is non-trivial. It requires a deep understanding about the structure of ELF binaries and significant amounts of engineering efforts. However, existing research [13] has shown that it is feasible. In this study, we investigate 5 modification methods and examine their impacts on end-to-end deep-learning malware detection. These 5 modification methods can modify a (relatively) large number of bytes and are general, as each one can be applied to most (if not all) ELF binaries created using a standard compiler (`gcc`, `clang`, etc). For each modification method we examine, we manually validate that a modified ELF binary remains functional.


Padding bytes (6 bytes, all 0x00)
Original ELF (sections)
Section offset 0x00000542,
Section Size 0x0010
Next section offset 0x00000558
Original ELF (Section Header Table)
Padding bytes (6 bytes, all 0xFF)
Modified ELF (sections)

```
00000520  73 74 61 72 74 5F 5F 00 5F 49 54 4D 5F 72 65 67  start__._ITM_reg
00000530  69 73 74 65 72 54 4D 43 6C 6F 6E 65 54 61 62 6C  isterTMCloneTabl
00000540  65 00 00 00 02 00 01 00 03 00 04 00 01 00 01 00  e...............
00000550  04 00 00 00 00 00 00 00 01 00 03 00 3A 00 00 00  ............:...
00000560  10 00 00 00 00 00 00 00 75 1A 69 09 00 00 04 00  ........u.i.....
00000570  44 00 00 00 10 00 00 00 14 69 69 0D 00 00 03 00  D........ii.....
```

```
00003900  00 00 00 00 00 00 00 00 71 00 00 00 FF FF FF 6F  ........q......o
00003910  02 00 00 00 00 00 00 00 42 05 00 00 00 00 00 00  ........B.......
00003920  42 05 00 00 00 00 00 00 10 00 00 00 00 00 00 00  B...............
00003930  06 00 00 00 00 00 00 00 02 00 00 00 00 00 00 00  ................
00003940  02 00 00 00 00 00 00 00 7E 00 00 00 FE FF FF 6F  ........~......o
00003950  02 00 00 00 00 00 00 00 58 05 00 00 00 00 00 00  ........X.......
00003960  58 05 00 00 00 00 00 00 40 00 00 00 00 00 00 00  X.......@.......
00003970  07 00 00 00 01 00 00 00 08 00 00 00 00 00 00 00  ................
```

```
00000520  73 74 61 72 74 5F 5F 00 5F 49 54 4D 5F 72 65 67  start__._ITM_reg
00000530  69 73 74 65 72 54 4D 43 6C 6F 6E 65 54 61 62 6C  isterTMCloneTabl
00000540  65 00 00 00 02 00 01 00 03 00 04 00 01 00 01 00  e...............
00000550  04 00 FF FF FF FF FF FF 01 00 03 00 3A 00 00 00  ............:...
00000560  10 00 00 00 00 00 00 00 75 1A 69 09 00 00 04 00  ........u.i.....
00000570  44 00 00 00 10 00 00 00 14 69 69 0D 00 00 03 00  D........ii.....
```

Fig. 5: An example of Padding Alteration: all the 6 padding bytes are modified to `0xFF`.

It is worth mentioning that our list of modification methods on ELF binaries is obviously not complete. There are more comprehensive modification methods that could perturb bytes in ELF binaries, especially when analyzing each ELF binary individually.

Header Alteration (HA): Header Alteration can modify bytes in the header of an ELF binary. Specifically, `e_flags` has 4 bytes (all 0x00 by default) that can be freely modified. In addition, there are 9 padding bytes (`EI_PAD`) in the ELF identification portion that can be modified without affecting functionalities. Overall, there are up to 13 modifiable bytes using Header Alteration. An example of Header Alteration is illustrated in Fig. 3.

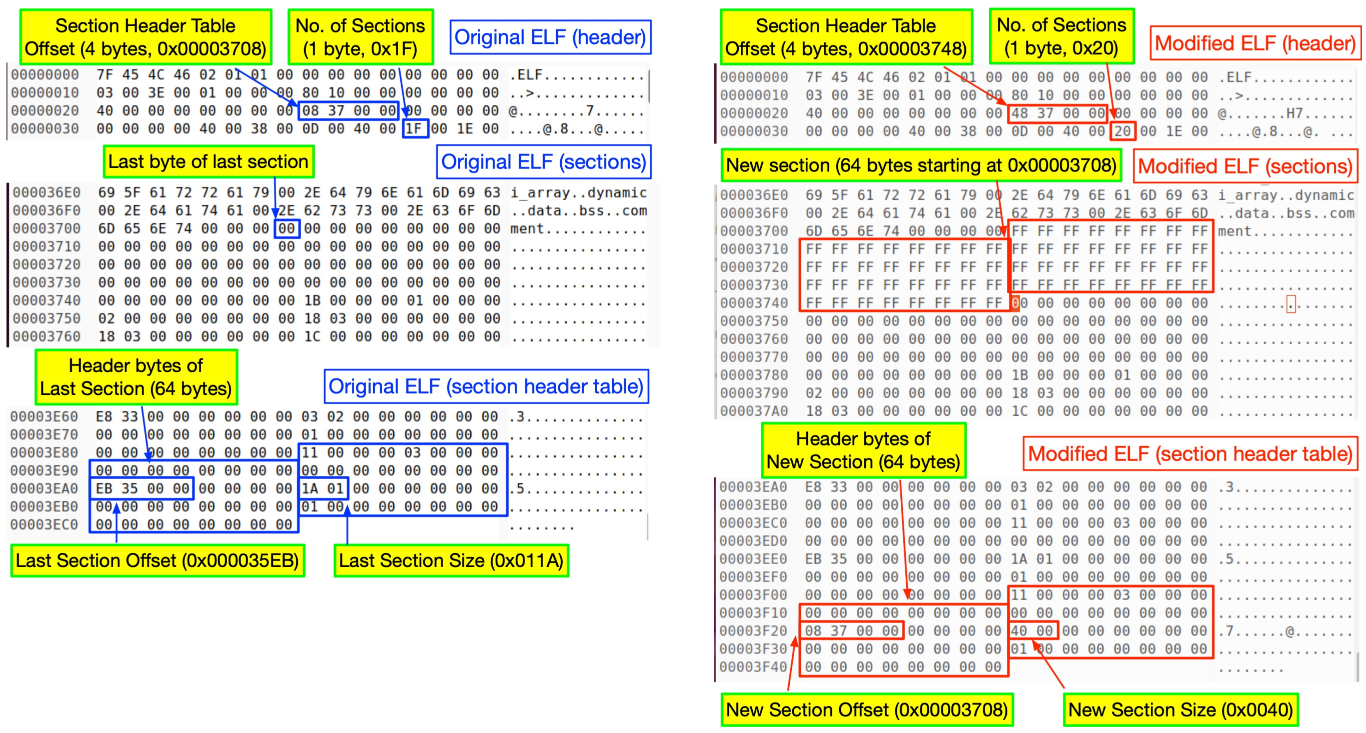
Debug Alteration (DA): Debug Alteration can modify bytes in multiple sections, including ".comment",".note", and ".debug" sections, where these sections include version control, vendor compliance, and debugging information respectively. All the bytes in these 3 sections can be modified without affecting the execution, but the particular number of bytes varies across ELF binaries. In general, it is around 50-300 bytes (in total) in one ELF binary. It is worth mentioning that these sections are available in non-stripped binaries but not in stripped binaries[1]. We assume all the ELF binaries in this study are non-stripped, which is the default in practice[2].

An example of Debug Alteration applied to a .comment section is presented in Fig. 4. The specific steps for generating the modified ELF binary are described below.

- We first scan the ELF Header to find the string table index. Given the index, we scan the Section Header Table to find the offset of the string table. From the string table, we can

[1]To generate a stripped binary, one can use `-s` option when compile the C code with `gcc` or use `strip` command on a non-stripped binary.

[2]Modifying bytes in stripped binaries is extremely challenging. Although it is an interesting problem, it is out of the scope of this study.


Section Header Table Offset (4 bytes, 0x00003708)
No. of Sections (1 byte, 0x1F)
Original ELF (header)
Last byte of last section
Original ELF (sections)
Header bytes of Last Section (64 bytes)
Original ELF (section header table)
Last Section Offset (0x000035EB)
Last Section Size (0x011A)
Section Header Table Offset (4 bytes, 0x00003748)
No. of Sections (1 byte, 0x20)
Modified ELF (header)
New section (64 bytes starting at 0x00003708)
Modified ELF (sections)
Header bytes of New Section (64 bytes)
Modified ELF (section header table)
New Section Offset (0x00003708)
New Section Size (0x0040)

```
00000000  7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00 .ELF............
00000010  03 00 3E 00 01 00 00 00 80 10 00 00 00 00 00 00 ..>.............
00000020  40 00 00 00 00 00 00 00 08 37 00 00 00 00 00 00 @........7......
00000030  00 00 00 00 40 00 38 00 0D 00 40 00 1F 00 1E 00 ....@.8...@.....
```

```
000036E0  69 5F 61 72 72 61 79 00 2E 64 79 6E 61 6D 69 63 i_array..dynamic
000036F0  00 2E 64 61 74 61 00 2E 62 73 73 00 2E 63 6F 6D ..data..bss..com
00003700  6D 65 6E 74 00 00 00 00 00 00 00 00 00 00 00 00 ment............
00003710  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00003720  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00003730  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00003740  00 00 00 00 00 00 00 00 1B 00 00 00 01 00 00 00 ................
00003750  02 00 00 00 00 00 00 00 18 03 00 00 00 00 00 00 ................
00003760  18 03 00 00 00 00 00 00 1C 00 00 00 00 00 00 00 ................
```

```
00003E60  E8 33 00 00 00 00 00 00 03 02 00 00 00 00 00 00 .3..............
00003E70  00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 ................
00003E80  00 00 00 00 00 00 00 00 11 00 00 00 03 00 00 00 ................
00003E90  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00003EA0  EB 35 00 00 00 00 00 00 1A 01 00 00 00 00 00 00 .5..............
00003EB0  00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 ................
00003EC0  00 00 00 00 00 00 00 00                         ........
```

```
00000000  7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00 .ELF............
00000010  03 00 3E 00 01 00 00 00 80 10 00 00 00 00 00 00 ..>.............
00000020  40 00 00 00 00 00 00 00 48 37 00 00 00 00 00 00 @.......H7......
00000030  00 00 00 00 40 00 38 00 0D 00 40 00 20 00 1E 00 ....@.8...@. ...
```

```
000036E0  69 5F 61 72 72 61 79 00 2E 64 79 6E 61 6D 69 63 i_array..dynamic
000036F0  00 2E 64 61 74 61 00 2E 62 73 73 00 2E 63 6F 6D ..data..bss..com
00003700  6D 65 6E 74 00 00 00 00 FF FF FF FF FF FF FF FF ment............
00003710  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
00003720  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
00003730  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
00003740  FF FF FF FF FF FF FF FF 00 00 00 00 00 00 00 00 ................
00003750  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00003760  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00003770  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00003780  00 00 00 00 00 00 00 00 1B 00 00 00 01 00 00 00 ................
00003790  02 00 00 00 00 00 00 00 18 03 00 00 00 00 00 00 ................
000037A0  18 03 00 00 00 00 00 00 1C 00 00 00 00 00 00 00 ................
```

```
00003EA0  E8 33 00 00 00 00 00 00 03 02 00 00 00 00 00 00 .3..............
00003EB0  00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 ................
00003EC0  00 00 00 00 00 00 00 00 11 00 00 00 03 00 00 00 ................
00003ED0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00003EE0  EB 35 00 00 00 00 00 00 1A 01 00 00 00 00 00 00 .5..............
00003EF0  00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 ................
00003F00  00 00 00 00 00 00 00 00 11 00 00 00 03 00 00 00 ................
00003F10  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00003F20  08 37 00 00 00 00 00 00 40 00 00 00 00 00 00 00 .7......@.......
00003F30  00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 ................
00003F40  00 00 00 00 00 00 00 00                         ........
```

Fig. 6: An example of End Appendix: a new section with 64 dummy bytes is appended starting at offset 0x00003708.

learn the index of the .comment section (details skipped in Fig. 4).

- Given the index of the .comment section, we scan the Section Header Table to find the offset of the .comment section (0x00003010) and its size (45 or 0x002D).
- We modify 45 bytes starting from offset 0x00003010.

Padding Alteration (PA). Padding Alteration modifies padding bytes, which are dummy zero (0x00) bytes before the end of each section in an ELF binary. Specifically, if the number of bytes associated with program instructions in a section is not a multiple of the word size, a compiler will automatically append a minimal number of zero bytes to the end of a section. We denote these zero bytes as padding bytes. The offset and size of a section defined in the Section Header Table ensure these padding bytes are not involved in the program execution. As a result, padding bytes can be modified arbitrarily without affecting functionalities. The number of padding bytes at the end of a section can be computed based on the offset of this section, the size of this section, and the offset of the next section.

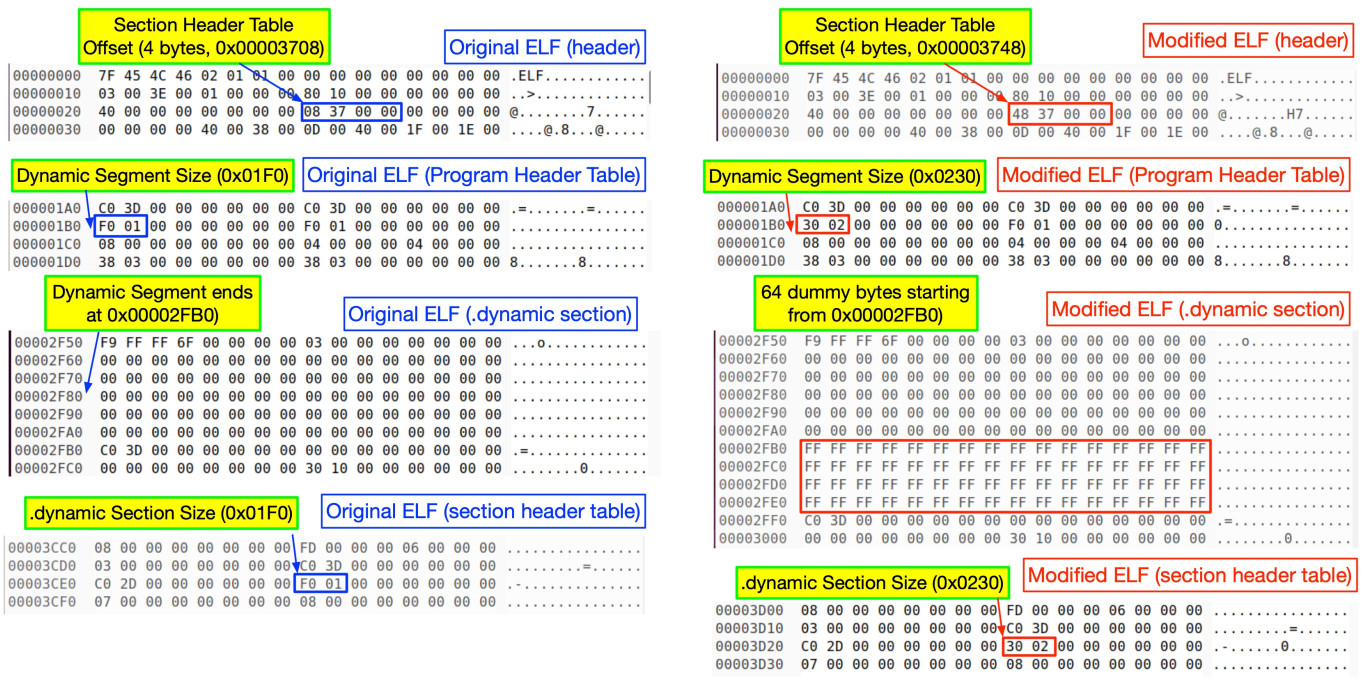
A concrete example of altering 6 bytes with Padding Alteration is illustrated in Fig. 5. The specific steps for generating the modified ELF binary are described below.

1) We pick a section and find its offset (0x00000542) and size (0x0010) in the Section Header Table.
2) We find the next section offset (0x00000558) in the Section Header Table.
3) We calculate the number of padding bytes in the section as 0x0558 - 0x0542 - 0x0010 (i.e., 1368 - 1346 - 16) = 6.
4) We modify the 6 padding bytes before offset 0x00000558.

End Appendix (EA). End Appendix can append a new section with an *arbitrary* number of dummy bytes near the end of the file (i.e., after the last section but before the Section Header Table). Specifically, given an original ELF binary, we first decide how many bytes we need to include in this new section. Then, we modify multiple bytes associated with this new section in the ELF Header (including the number of sections and the offset of Section Header Table) and also multiple bytes in the Section Header Table (including the size and offset of this new section) . Next, the new section with dummy bytes is appended to the last section of the original binary. These dummy bytes can be arbitrary bytes. Particularly, for ease of implementation, we implement two options: (1) constant bytes (e.g., all 0xFF) or (2) variable bytes (e.g., bytes from benign binaries).

A concrete example of appending 64 bytes with End Appendix is illustrated in Fig. 6. The specific steps for generating the modified ELF binary are described below.

1) We choose to create a new section with 64 dummy bytes, where each byte is 0xFF.
2) We record the Section Header Table offset (0x00003708) in the original ELF Header and leverage this offset as the offset of the new section.
3) All the bytes starting at 0x00003708 in the original ELF binary are shifted down with an offset of 64 (i.e., 0x40), which is the number of dummy bytes. As a result, we need to increase the Section Header Table offset by 64 (0x00003708 to 0x00003748) and increase the


Section Header Table Offset (4 bytes, 0x00003708)
Original ELF (header)
Dynamic Segment Size (0x01F0)
Original ELF (Program Header Table)
Dynamic Segment ends at 0x00002FB0)
Original ELF (.dynamic section)
.dynamic Section Size (0x01F0)
Original ELF (section header table)
Section Header Table Offset (4 bytes, 0x00003748)
Modified ELF (header)
Dynamic Segment Size (0x0230)
Modified ELF (Program Header Table)
64 dummy bytes starting from 0x00002FB0)
Modified ELF (.dynamic section)
.dynamic Section Size (0x0230)
Modified ELF (section header table)

```
00000000  7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00 .ELF............
00000010  03 00 3E 00 01 00 00 00 80 10 00 00 00 00 00 00 ..>.............
00000020  40 00 00 00 00 00 00 00 08 37 00 00 00 00 00 00 @........7......
00000030  00 00 00 00 40 00 38 00 0D 00 40 00 1F 00 1E 00 ....@.8...@.....
```

```
000001A0  C0 3D 00 00 00 00 00 00 C0 3D 00 00 00 00 00 00 .=.......=......
000001B0  F0 01 00 00 00 00 00 00 F0 01 00 00 00 00 00 00 ................
000001C0  08 00 00 00 00 00 00 00 04 00 00 00 04 00 00 00 ................
000001D0  38 03 00 00 00 00 00 00 38 03 00 00 00 00 00 00 8.......8.......
```

```
00002F50  F9 FF FF 6F 00 00 00 00 03 00 00 00 00 00 00 00 ...o............
00002F60  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00002F70  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00002F80  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00002F90  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00002FA0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00002FB0  C0 3D 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .=..............
00002FC0  00 00 00 00 00 00 00 00 30 10 00 00 00 00 00 00 ........0.......
```

```
00003CC0  08 00 00 00 00 00 00 00 FD 00 00 00 06 00 00 00 ................
00003CD0  03 00 00 00 00 00 00 00 C0 3D 00 00 00 00 00 00 .........=......
00003CE0  C0 2D 00 00 00 00 00 00 F0 01 00 00 00 00 00 00 .-..............
00003CF0  07 00 00 00 00 00 00 00 08 00 00 00 00 00 00 00 ................
```

```
00000000  7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00 .ELF............
00000010  03 00 3E 00 01 00 00 00 80 10 00 00 00 00 00 00 ..>.............
00000020  40 00 00 00 00 00 00 00 48 37 00 00 00 00 00 00 @.......H7......
00000030  00 00 00 00 40 00 38 00 0D 00 40 00 1F 00 1E 00 ....@.8...@.....
```

```
000001A0  C0 3D 00 00 00 00 00 00 C0 3D 00 00 00 00 00 00 .=.......=......
000001B0  30 02 00 00 00 00 00 00 F0 01 00 00 00 00 00 00 0...............
000001C0  08 00 00 00 00 00 00 00 04 00 00 00 04 00 00 00 ................
000001D0  38 03 00 00 00 00 00 00 38 03 00 00 00 00 00 00 8.......8.......
```

```
00002F50  F9 FF FF 6F 00 00 00 00 03 00 00 00 00 00 00 00 ...o............
00002F60  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00002F70  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00002F80  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00002F90  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00002FA0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00002FB0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
00002FC0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
00002FD0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
00002FE0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
00002FF0  C0 3D 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .=..............
00003000  00 00 00 00 00 00 00 00 30 10 00 00 00 00 00 00 ........0.......
```

```
00003D00  08 00 00 00 00 00 00 00 FD 00 00 00 06 00 00 00 ................
00003D10  03 00 00 00 00 00 00 00 C0 3D 00 00 00 00 00 00 .........=......
00003D20  C0 2D 00 00 00 00 00 00 30 02 00 00 00 00 00 00 .-......0.......
00003D30  07 00 00 00 00 00 00 00 08 00 00 00 00 00 00 00 ................
```

Fig. 7: An example of Dynamic Extension: 64 dummy bytes are inserted starting at offset `0x00002FB0`.

number of sections by 1 (`0x1F` to `0x20`) in the ELF Header.

4) We insert the new section with 64 dummy bytes starting at `0x00003708`
5) We create 64 header bytes for the new section at the end of the Section Header Table. We add these 64 bytes by copying the 64 header bytes from the last section of the original ELF binary but assigning the new section offset as `0x00003708` and the new section size as `0x0040`.

Dynamic Extension (DE). Dynamic Extension extends the size of the dynamic segment (i.e., PT_DYNAMIC) by appending dummy bytes at the end of it. The dynamic segment specifies dynamic linking information for an ELF binary and appears (relatively) late in an ELF binary, typically after most PT_LOAD segments. The dynamic segment contains only one section, named the .dynamic section. Since the bytes of the dynamic segment are not loaded into memory but only read by the dynamic linker during execution, an arbitrary number of dummy bytes can be appended at the end of the dynamic segment without affecting execution.

To append bytes successfully, associated information regarding the size of this segment, the size of the .dynamic section, the offsets of all the subsequent segments and sections, and the offset of the Section Header Table will all need to be updated in the ELF Header, the Program Header Table, and the Section Header Table respectively. This modification shares a similar concept as End Appendix but modifies bytes earlier in a binary. The main difference is that this modification also needs to modify bytes in the Program Header Table while End Appendix does not.

A concrete example of adding 64 bytes in the dynamic segment with Dynamic Extension is illustrated in Fig. 7. The specific steps for generating the modified ELF binary are described below.

1) We choose to extend the dynamic segment (i.e., .dynamic section) with 64 dummy bytes, where each byte is `0xFF`.
2) We find the offset of the segment,`0x00002FB0`, next to the dynamic segment by scanning each segment defined in the Program Header Table (skipped in Fig. 7).
3) We increase the offset of all bytes starting at `0x00002FB0` in the original ELF binary by 64 (0x40), which is the number of dummy bytes.
4) We insert 64 dummy bytes starting at `0x00002FB0`.
5) We increase the size of the dynamic segment in the Program Header Table by 64 (`0x10F0` to `0x2030`). We increase the size of the .dynamic section in the Section Header Table by 64 (`0x10F0` to `0x2030`).
6) We increase the offset of the Section Header Table by 64 (`0x00003708` to `0x00003748`).
7) Besides the Section Header Table, if there are more sections or segments that are after the dynamic segment, we increase the offset of each of these sections by 64 in the Section Header Table and increase the offset of each of these segments by 64 in the Program Header Table (skipped in Fig. 7).

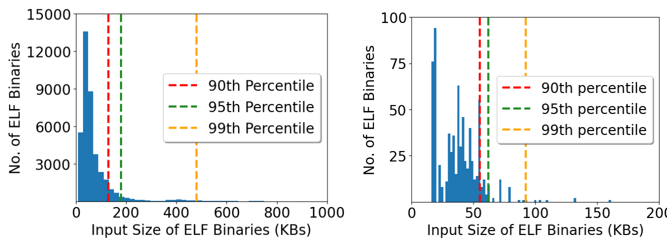
V. Evaluation

Dataset. We leverage a public dataset[3], referred to as the *Labeled-Elfs* dataset. It contains a total of 39,521 ELF binaries (711 malicious binaries and 38,810 benign binaries) produced using x86-64 architecture (little endian)[4]. The 711 malicious

[3]https://github.com/nimrodpar/Labeled-Elfs

[4]There is also a small number of benign binaries generated for ARM 32 in the original dataset, we exclude those in our study.

ELF binaries were produced from 4 malware (written in C), including Mirai-vanilla, BASHLITE-1.0, BASHLITE-lizkebab, and lightaidra-1.0. Multiple compilers (`gcc`, `clang`, and `llvm`) with different versions and various optimization levels (O1, O2, O3, and Os) were applied when producing these binaries. All the ELF binaries are non-stripped, and some of the malware binaries are obfuscated[5]. The distribution of file size of this dataset is presented in Fig. 8.



(a) All Binaries (b) Malicious Binaries Only

Fig. 8: Distribution of ELF binary size (*Labeled-Elfs*).

We establish one subset from this dataset for our evaluation. We refer this subset as *Labeled-Elfs-Balanced*. It contains all the 711 malicious binaries and 711 random benign binaries from the original dataset.

Neural Networks. We use two CNNs, including MalConv and FireEyeNet, as baseline detectors that a black-box adversary could attack. Both networks are originally designed for deep-learning malware detection over PE files in Windows.

MalConv. MalConv [1] is a neural network that combines a convolutional neural network with a global max-pooling before transferring to connected layers. This model uses one 8-dimensional embedding layer, two 1-dimensional gated convolutional layers, a temporal max pooling layer, and a fully connected layer with softmax. The embedding layer maps each byte to a fixed length feature vector, which reduces bias in byte values. Also, the convolutional layers holds a large filter width of 500 bytes and a stride of 500 bytes, with 128 filters total. The maximum input size examined in [1] is 1 MB. We set window size as 500, epochs as 50, batch size as 32, and learning rate as 0.0001.

FireEyeNet. FireEyeNet [2] was proposed by researchers from FireEye. It consists of one 10-dimensional embedding layer, five stacked 1-dimensional convolutional and max pooling layers, followed by a fully connected layer with sigmoid function. The maximum input size of each program examined in [2] is 102,400 and it achieves 98% AUC and 96% accuracy over a private large-scale PE binary dataset.

Experiment Setting. We use a Linux machine with an i5 CPU, 32GB memory, and one Nvidia Titan RTX GPU to perform all the experiments. We develop a tool, named *EvilELF*, to perform each modification automatically over an ELF binary.

Experiment 1: Performance of Baseline Detectors. We investigate the performance of baseline detectors for end-to-end deep-learning malware detection over ELF files. Specifically, we leverage MalConv as the architecture of baseline neural networks and we explore multiple detectors with various input sizes, including 100K, 200K, 500K, and 1000K respectively, by using Labeled-Elfs-Balanced.

When we train each baseline detector, we use 80% of data for training, 10% for validation, and 10% for testing. Detailed results are presented in Table I. In addition to MalConv, we also train baseline detector with FireEyeNet using input size 102400, 204800, and 409600 respectively. *Overall, we observe that these baseline detectors have promising performance in malware detection.*

TABLE I: The performance of baseline detectors over Labeled-Elfs-Balanced

Detector	ACC	Precision	Recall	F1	AUC
MalConv_100k	97.2%	98.5%	95,8%	97.2%	99.5%
MalConv_200k	97.2%	100.0%	100.0%	97.2%	99.5%
MalConv_500k	97.9%	98.5%	92.1%	95.2%	99.6%
MalConv_1000k	95.1%	97.1%	95.8%	96.5%	99.7%
FireEye_102400	98.5%	97.2%	97.1%	97.1%	99.5%
FireEye_204800	99.2%	100.0%	100.0%	99.3%	99.5%
FireEye_409600	97.8%	95.7%	95.8%	97.8%	98.7%

Experiment 2: Evasion Attacks on Baseline Detectors. We examine evasion attacks on baseline detectors. We apply each of the 5 modification methods separately. We examine both MalConv and FireEyeNet with different input sizes.

Specifically, given each modification method, we randomly pick 5 malware binaries that are predicted as malicious by all 4 MalConv baseline detectors, Then, we generate 1,000 modified versions of these 5 malware binaries by using the given modification method. Next, we pass these 1,000 modified binaries to each baseline detector to measure the evasion rate. We repeat the attacks with 5 trials and record the mean evasion rate. All the 1,000 modified binaries are different across the 5 trials. We repeat the same process for FireEyeNet detectors.

For End Appendix or Dynamic Extension, we examine two ways, including (1) constant dummy bytes and (2) variable dummy bytes from benign binaries, for altering bytes. For constant dummy bytes, we randomly choose a byte from [`0x40`, `0xFF`]. We set the number of dummy bytes as 200,000 in this experiment.

Observations from Experiment 2. As shown in Table II, we have 3 major observations

- *End Appendix (Constant), Dynamic Extension (Constant), and Padding Alteration are able to defeat baseline detectors.* For example, Padding Alteration achieves 8.4% evasion rate on MalConv_100k and 76.6% evasion rate on FireEye_102400.
- We also observe that *the evasion rate decreases when the input size of MalConv increases*. This suggests that MalConv with a higher input size is more resilient against evasion attacks with our modifications. Conversely, we observe FireEyeNet is more vulnerable with a larger input.
- *Header Alteration and Debug Alteration are ineffective (i.e., evasion rate is or is close to 0%).* This is likely

[5]Our modification methods also work for obfuscated non-stripped binaries.

TABLE II: Evasion Rate (mean) on Baseline Detectors

Detector	Header Alteration	Padding Alteration	Debug Alteration	End Appendix (Constant, 200k)	End Appendix (Variable, 200k)	Dynamic Extension (Constant, 200k)	Dynamic Extension (Variable, 200k)
MalConv_100k	0%	**8.4%**	0%	7.3%	0%	8.1%	0%
MalConv_200k	0%	**5.3%**	0%	4.5%	0.5%	4.8%	1.1%
MalConv_500k	0%	**2.6%**	0%	1.1%	0.2%	1.0%	0%
MalConv_1000k	0%	**1.6%**	0%	0.8%	0%	0.8%	0.1%
FireEye_102400	0%	**76.6%**	0.7%	42.5%	0.7%	51.9%	52.4%
FireEye_204800	0%	27.3%	2.0%	32.1%	2.0%	**31.7%**	31.1%
FireEye_409600	0%	20.2%	0.1%	48.8%	0.1%	73.9%	**74.6%**

TABLE III: The Impact of the Number of Dummy Bytes in End Appendix on Evasion Rate (mean)

	Constant Bytes					Variable Bytes (Benign)		
	100	1k	10k	100k	200k	100k	200k	400k
MalConv_100k	0%	7.3%	7.2%	7.3%	7.3%	0%	0%	0%
MalConv_200k	0%	4.5%	4.5%	4.5%	4.5%	0.1%	0.4%	0.4%
MalConv_500k	0%	1.1%	1.1%	1.1%	1.1%	0%	0.1%	0.2%
MalConv_1000k	0%	0.8%	0.8%	0.8%	0.8%	0%	0%	0.1%
FireEye_102400	2.2%	21.4%	24.0%	42.5%	42.5%	0.7%	0.7%	0.7%
FireEye_204800	15.7%	27.7%	28.2%	32.8%	32.1%	2.0%	2.0%	2.0%
FireEye_409600	2.2%	20.5%	20.5%	23.0%	48.9%	0.1%	0.1%	0.1%

TABLE IV: The Impact of the Number of Dummy Bytes in Dynamic Extension on Evasion Rate (mean)

	Constant Bytes					Variable Bytes (Benign)		
	100	1k	10k	100k	200k	100k	200k	400k
MalConv_100k	0%	7.2%	7.2%	8.1%	8.1%	0%	0%	0%
MalConv_200k	0%	3.3%	3.3%	4.6%	4.8%	0.1%	1.1%	1.1%
MalConv_500k	0%	0.9%	0.9%	1.0%	1.0%	0%	0%	0.1%
MalConv_1000k	0%	0.5%	0.5%	0.8%	0.8%	0%	0.1%	0.1%
FireEye_102400	6.4%	29.3%	33.4%	51.9%	51.9%	52.4%	52.4%	52.4%
FireEye_204800	13.5%	27.5%	27.9%	34.7%	31.7%	31.1%	31.1%	31.1%
FireEye_409600	58.6%	21.3%	20.2%	41.6%	74.0%	74.6%	74.6%	74.6%

because the two methods modify a small number of bytes.

- *MalConv is much more robust than FireEyeNet under evasion attacks.*

Experiment 3: Impacts of the Number of Dummy Bytes (End Appendix and Dynamic Extension). We examine the impacts of the number of dummy bytes in End Appendix and Dynamic Extension. Specifically, given a modification method, we again produce 1,000 malicious binaries and pass them to a baseline detector to measure the evasion rate. We investigate the impact of different amounts of dummy bytes.

As presented in Table III, we notice that if the number of dummy bytes is greater than 1,000, increasing the number of dummy bytes further in End Appendix (with constant bytes) does not affect the evasion rate for MalConv. We have a consistent observation for Dynamic Extension in Table IV. For FireEyeNet, the evasion rate continues to increase with the number of dummy bytes for constant byte modifications.

Experiment 4: Mitigating Evasion Attacks. In previous experiments, we have shown that MalConv with an input size of 1 million bytes is resillient under evasion attacks. In this experiment, we retrain a baseline detector by using the original binaries and also perturbed malicious binaries in order to reduce evasion rate in evasion attacks.

Specifically, given 711 malicious ELF binaries and 711 benign ELF binaries, we first generate 1,250 modified malicious binaries given each modification by following the steps in previous experiments. Then, we retrain a detector with 961 malicious binaries (711 original and 250 modified) and 711 benign binaries and measure the evasion rate on the retrained detector with the remaining 1,000 modified malicious binaries. We evaluate both MalConv and FireEyeNet with different input size. We find that retraining a neural network with deliberately modified binaries can effectively mitigate evasion attacks as shown in Table V. On the other hand, generating sufficient deliberately modified binaries with various modification methods could be difficult to scale, especially when there are a large number of malicious binaries in a dataset.

TABLE V: Evasion rate on detectors that are retrained with modified malicious binaries)

Detector	Padding Alteration
MalConv_100k	0%
MalConv_200k	0.2%
MalConv_500k	0%
MalConv_1000k	0%
FireEye_102400	3.1%
FireEye_204800	2.0%
FireEye_409600	1.0%

Experiment 5: Evading Real-World Malware Detectors on VirusTotal. We investigate whether our deliberately modified binaries could evade real-world malware detectors that may (or may not) use end-to-end malware detection. Specifically, we choose five original malicious binaries, and for each one, we generate one modified malicious binaries using Padding Alteration. Then, we submit the 5 original

TABLE VI: Evasion on Real-World Malware Detectors (VirusTotal, 62 detectors in total, examined in August 2023)

	Name of Malware Binary	No. of Detectors Reporting Malware
Original	lightaidra-1.0 (clang-6.0.1, Os)	23
	BASHLITE-client-1.0 (gcc-7.1.0, O0)	35
	BASHLITE-client-1.0 (gcc-9.1.0, O2)	29
	Mirai-vanilla (gcc-8.4.0, O0)	35
	Mirai-vanilla (gcc-8.4.0, Os)	9
Modified	lightaidra-1.0 (clang-6.0.1, Os)	**7**
	BASHLITE-client-1.0 (gcc-7.1.0, O0)	**33**
	BASHLITE-client-1.0 (gcc-9.1.0, O2)	**17**
	Mirai-vanilla (gcc-8.4.0, O0)	**7**
	Mirai-vanilla (gcc-8.4.0, Os)	**8**

malicious binaries and the 5 modified ones to VirusTotal[6], an online virus detection website. For each binary, VirusTotal returns detection results, either *malicious* or *benign*, from 62 major malware detection services, including Avast, Microsoft, Kaspersky, McAfee, etc.

We observe that (1) ***Each original malicious ELF binaries can be detected by about half of the real-world malware detectors;*** (2) ***The number of detectors that can detect each modified malicious ELF binary drops significantly.***

For instance, 35 detectors can label the original binary of Mirai-vanilla (compiled with 8.4.0 with O0 optimization). However, after our modification with Padding Alteration, only 7 detectors can still identify the modified one as malicious. It is also worth mentioning that, among all the 62 detectors, only 6 detectors, including AVG, Kaspersky, Avast, Microsoft, ZoneAlarm, and ESET-NOD32, are able to detect all the 5 modified binaries in this experiment.

VI. Discussion and Future Work

Combining Multiple Modification. We only investigate the cases where modified binaries are generated by a single modification method. Combinations of multiple modification methods can alter more bytes in a binary, and therefore, may lead to a higher evasion rate. We will leave this as future work.

More Modification Methods. There are other methods that can also modify ELF binaries. For instance, *patching* [15] is often used to modify specific instructions in a binary. It would be interesting to explore whether evasion attacks with patching are effective and to what degree. On the other hand, the bytes that can be modified with patching are specific in each binary while the methods we examine are generic.

Modifying Benign Binaries. We only examine modifications over malicious binaries in this study. An attack can also modify benign binaries such that it can include malicious instructions. It is even more challenging to achieve, especially with a great number of bytes. We leave it as future work.

Static Analysis Only. We demonstrate that our evasion attacks are effective on detectors based on static analysis. On the other hand, we acknowledge that our modified binaries cannot bypass detectors based on dynamic analysis (e.g., API calls) as our modifications do not change program execution.

[6]https://www.virustotal.com/gui/home/upload

Larger Datasets. We use a dataset with less than 1,500 binaries in our evaluations. Having a larger dataset and observing the results over it would be interesting. However, large-scale datasets with malware ELF binaries are often not publicly available or difficult to acquire.

VII. Conclusion

We examine five modification methods that can generate malicious ELF binaries without affecting original functionalities. In addition, we leverage these modifications in evasion attacks on end-to-end deep learning malware detection. Experimental results show that evasion attacks on end-to-end deep learning malware detection is feasible. We also observe that retraining malware detectors with deliberately modified malicious binaries can significantly mitigate evasion attacks.

Acknowledgements

This work was partially supported by National Science Foundation (CNS-2150086) and Air Force Research Laboratory (Contract No. AFRL-2021-1805).

References

[1] E. Raff, J. Baker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nickolas, "Malware detection by eating a whole exe," in *2018 AAAI Workshops on AI for Cybersecurity*, 2018.

[2] S. E. Coull and C. Gardner, "Activation Analysis of a Byte-Based Deep Neural Network for Malware Classification," in *2nd Deep Learning and Security Workshop*, 2019.

[3] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli, "Adversarial malware binaries: Evading deep learning for malware detection in executables," *2018 26th European Signal Processing Conference (EUSIPCO)*, pp. 533–537, 2018.

[4] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando, "Explaining vulnerabilities of deep learning to adversarial malware binaries," in *CEUR Workshop Proceedings*, Pisa, Italy, 2019.

[5] F. Kreuk, A. Barak, S. Aviv-Reuven, M. Baruch, B. Pinkas, and J. Keshet, "Deceiving end-to-end deep learning malware detectors using adversarial examples," in *Workshop on Security in Machine Learning (NeurIPS)*, 2018.

[6] O. Suciu, S. E. Coull, and J. Johns, "Exploring adversarial examples in malware detection," *2019 IEEE Security and Privacy Workshops (SPW)*, 2019.

[7] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando, "Functionality-preserving black-box optimization of adversarial windows malware," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 3469–3478, 2021.

[8] A. Khormali, A. Abusnaina, S. Chen, D. Nyang, and A. Mohaisen, "COPYCAT: Practical Adversarial Attacks on Visualization-Based Malware Detection," https://arxiv.org/abs/1909.09735.

[9] H. Anderson, A. Kharkar, B. Filar, D. Evans, and P. Roth, "Learning to Evade Static PE Machine Learning Malware Models via Reinforcement Learning," https://arxiv.org/abs/1801.08917.

[10] K. Lucas, M. Sharif, L. Bauer, M. K. Reiter, and S. Shintre, "Malware Makeover: Breaking ML-based Static Analysis by Modifying Executable Bytes," in *Proc. of ACM ASIACCS'21*, May 2021.

[11] H. Liu, W. Sun, N. Niu, and B. Wang, "MultiEvasion: Evasion Attacks against Multiple Malware Detectors," in *Proc. of IEEE CNS'22*, 2022.

[12] L. Demetrio, S. E. Coull, B. Biggio, G. Lagorio, A. Armando, and F. Roli, "Adversarial EXEmples: A Survey and Experimental Evaluation of Practical Attacks on Machine Learning for Windows Malware Detection," *ACM Transactions on Privacy and Security*, vol. 24, no. 4, 2021.

[13] Y. Qiao, W. Zhang, Z. Tian, L. T. Yang, Y. Liu, and M. Alazab, "Adversarial ELF Malware Detection Method Using Model Interpretation," *IEEE Transactions on Industrial Informatics*, 2023.

[14] "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification," https://refspecs.linuxfoundation.org/elf/elf.pdf.

[15] R. O'Neill, *Learning Linux Binary Analysis*. Packt Publishing, 2016.