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 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, that 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 with 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 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 Alteration are not effective.
- We find MalConv is much more resilient than FireEye, where an attack achieves a much lower evasion rate. For instance, evasion rate is only up to 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 DOS header. Kreuk et al. [5] developed a gradient-based evasion attack that perturb 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] proposed

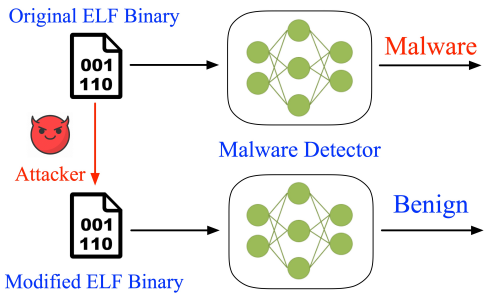


Fig. 1: System and threat model

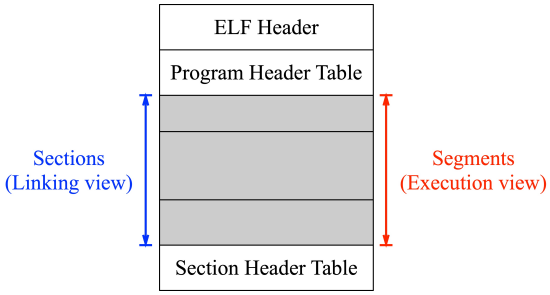


Fig. 2: The high-level structure of an ELF binary

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 .text section with semantic equivalent opcodes or uses *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 but no real-world modified binaries are generated.

Their second method inserts a new section between two sections by modifying section offsets in Section Header Table. However, offsets in Program Header Table are not modified correspondingly. As a result, the modification is not completed where a modified ELF binary may not be able 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 passing 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, Area Under the Curve (AUC) to measure the performance of a malware detector. In addition, we use ***evasion rate*** to measure to what degree the evasion attack is effective. 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 can be 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 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 enables the operating system to execute the binary by informing where the segments that need to be loaded into memory.

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

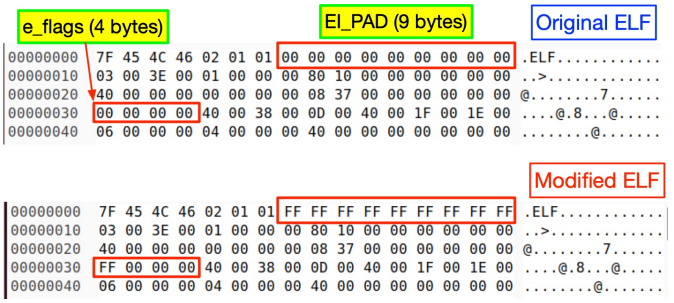


Fig. 3: An example of Header Alteration: all the 9 padding bytes and the first byte of `e_flags` are modified to `0xFF`.

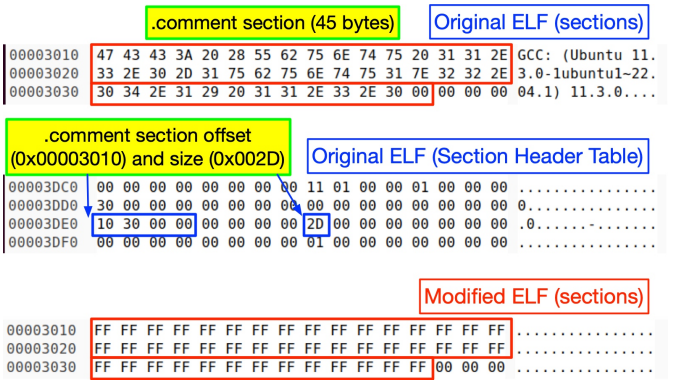


Fig. 4: An example of Debug Alteration: all the 45 bytes of `.comment` section are modified to `0xFF`.

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), read-only data (.rodata), executable instructions (.text), a string table (.strtab), and a symbol table (.symtab). A segment contains one or multiple sections. Common segments include 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 to ensure there are no overlapping among sections. It is not loaded during the program execution but 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] have shown that it is feasible. In this study, we particularly investigate 5 modification methods and examine their impacts to end-to-end deep-learning malware detection. These 5 modification methods can modify a (relatively) large number of bytes and are generic, where each one can be applied to most (if not all) ELF binaries created using a standard compiler (`gcc`, `clang`,

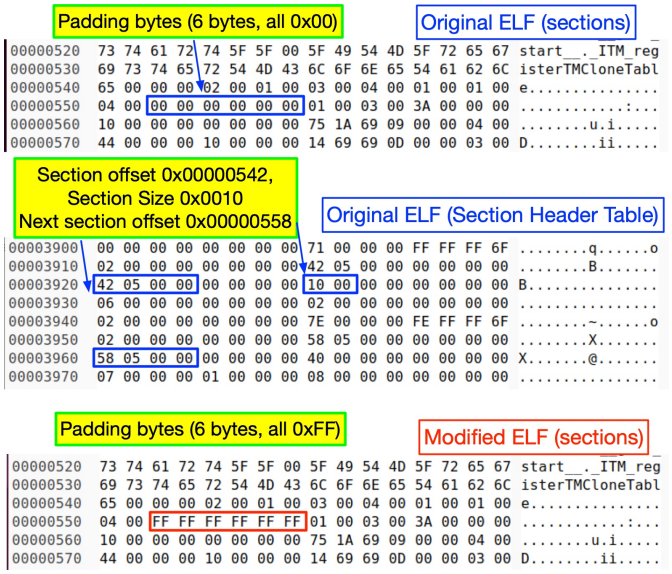


Fig. 5: An example of Padding Alteration: all the 6 padding bytes are modified to `0xFF`.

etc). For each modification method we examine, we manually validate an modified ELF binary remains functional.

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) and can be freely modified. In addition, there are 9 padding bytes (`EI_PAD`) in the ELF identification portion and can be modified without affecting functionalities. Overall, there are up to 13 bytes can be modified by 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", where these sections include version control, vendor compliance, and debugging information. All the bytes in these 3 sections can be modified without affecting the execution. The particular number of bytes included in these 3 sections of an ELF binary varies. 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 default in practice[2].

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

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

[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.

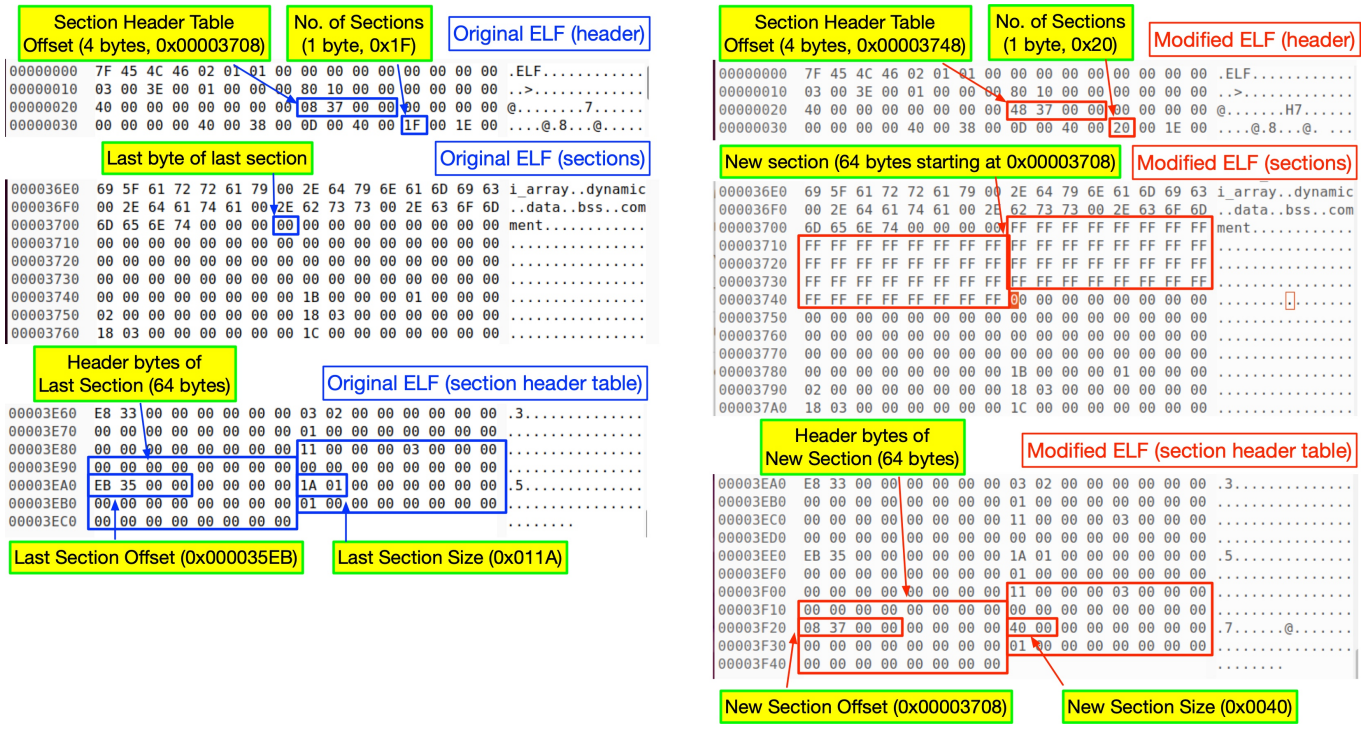


Fig. 6: An example of End Appendix: a new section with 64 dummy bytes is appended starting at offset `0x00003708`.

offset of string table. From the string table, we can learn the index of .comment section (details skipped in Fig. 4).
- Given the index of .comment section, we scan the Section Header Table to find the offset of .comment section (`0x00003010`) and its size 45 (`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 a 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 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 Section Header Table.
2) We find the next section offset (`0x00000558`) in 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 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 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 ELF Header (including the number of sections and the offset of Section Header Table) and also multiple bytes in 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 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 Section Header Table offset by 64 (`0x00003708` to `0x00003748`) and increase the number of sections by 1 (`0x1F` to `0x20`) in ELF Header.

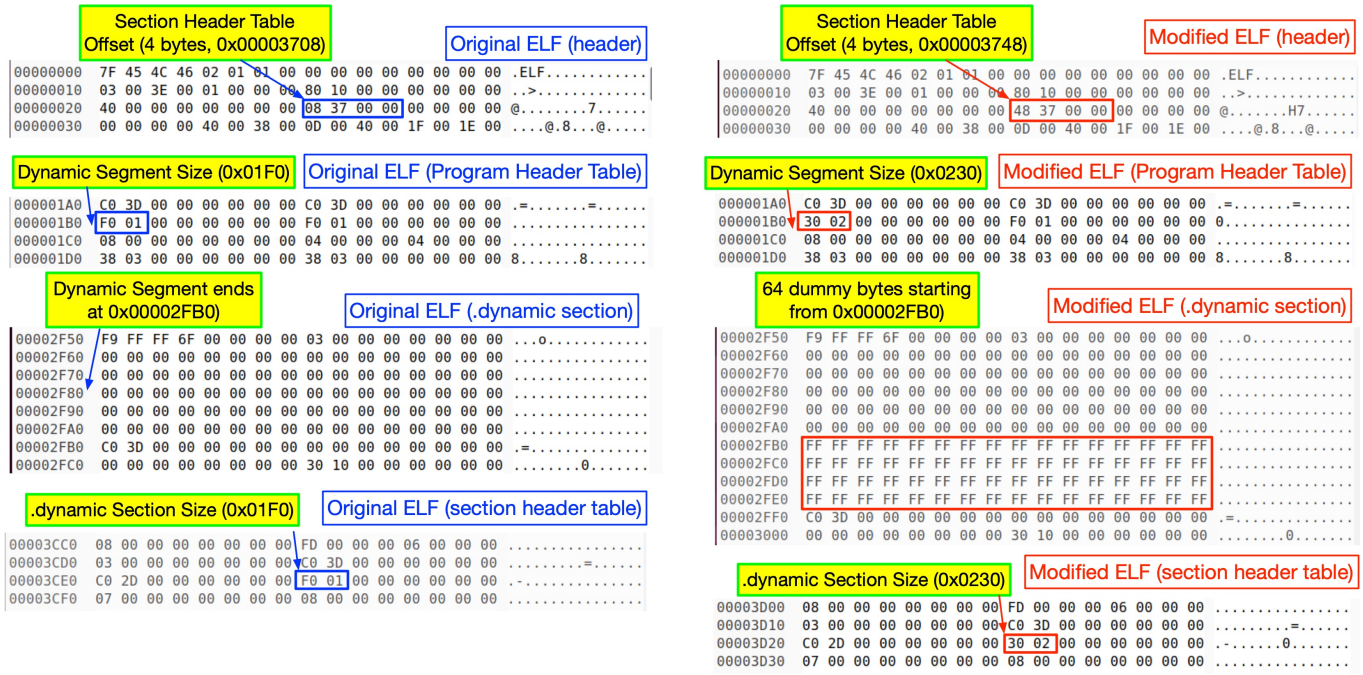


Fig. 7: An example of Dynamic Extension: 64 dummy bytes are inserted starting at offset `0x00002FB0`.

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

A concrete example of extending new 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, `0x00002FB0`, the offset of the segment next to the dynamic segment by scanning each segment defined in Program Header Table (skipped in Fig. 7).
3) We move all the bytes starting at `0x00002FB0` in the original ELF binary down with an offset of 64 (0x40), which is the number of dummy bytes.
4) We insert 64 dummy bytes starting at `0x00002FB0`.
5) We increase the size of dynamic segment in Program Header Table by 64 (`0x10F0` to `0x2030`). We increase the size of .dynamic section in Section Header Table by 64 (`0x10F0` to `0x2030`).
6) We increase the offset of Section Header Table by 64 (`0x00003708` to `0x00003748`).
7) Besides 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 Section Header Table and increase the offsset of each of these segment by 64 in Program Header Table (skipped in Fig. 7).

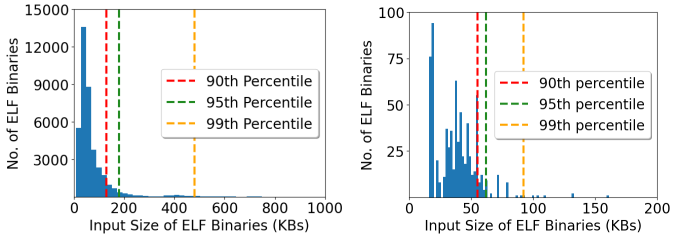
V. Evaluation

Dataset. We leverage a public dataset[3], referred to as *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 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

[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.

`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.

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 an 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 dataset.

Experiment Setting. We use a Linux machine with i5 CPU, 32GB memory, and one Nivida 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 the 4 MalConv baseline detectors, Then, we generate 1,000 modified binaries 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 Segment 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. For FireEyeNet, we observe that a greater input size may make the detector even more vulnerable under evasion attacks.
- *Header Alteration and Debug Alteration are not effective (i.e., evasion rate is 0% or close to 0%)*. This is likely because the two methods can only modify a small number of bytes.

[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%

- *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 still produce 1,000 malicious binaries as in previous experiment and pass them to a baseline detector to measure evasion rate. We investigate the different sizes of dummy bytes.

As presented in Table III, we notice that, when 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 evasion rate on MalConv. We have consistent observation about Dynamic Extension from Table IV. For FireEyeNet, the evasion rate increases when we increase the number of dummy bytes in End Appendix (constant bytes) or Dynamic Extension (constant bytes).

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 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, it is also worth mentioning that 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) built based on end-to-end detection over bytes. Specifically, we choose five original malicious binaries. For each original one, we generate one modified malicious binaries using Padding Alteration. Then, we submit the 5 orig-

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**

inal 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. The results of each binary are reported within a few seconds.

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 methods. Combination of multiple modification methods can alter more bytes in a binary, and therefore, could lead to 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

Greater 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.