

AI-Based Generation of Adversarial Malicious ELF Samples



Reporter: Heyi Wu

2024 /11/7

CONTENTS

01. OVERVIEW

02. TECHNICAL OBJECTIVES


03. TECHNICAL SOLUTIONS



PART ONE

OVERVIEW

1.1 Background Overview



Chinese government and enterprise organizations will replace domestic operating systems and application software on a large scale. However, most security vendors are currently unable to cover this area. Moreover, the domestic terminal security software has weak capability in detecting and killing malicious ELF samples. Some hackers and attack teams have targeted products that use domestic operating systems, and the existing detection capabilities are easily bypassed by attackers. Worse still, there is no established channel for exchanging malicious ELF samples among vendors. Thus, a system that generates adversarial malicious samples in the domestic environment using AI technology is needed to establish an advantage in the malicious sample library, and enhance the capability to detect and kill malicious software.

1.1 Background Overview

1. There are very few domestically produced ELF malicious sample databases

Currently, most of the work focuses on the research of PE malicious samples. Meanwhile, databases for malicious software in PE format have been established, such as EMBER. However, there is currently a blank for domestically produced ELF malicious sample databases. The development difficulty for ELF malicious samples is relatively high. Indeed, this project requires the ability to run ELF malicious software on the domestically produced OS platforms, which are based on x86/ARM architecture. Thus, the collection of ELF malicious samples is targeted and the development difficulty of the sample set is very high.

2. There is zero work on the generation of adversarial malicious software targeting ELF

In most of the published papers, more than 70 papers are related to PE malicious software detection, classification, and countermeasures against attacks, while only 15 papers are related to ELF malicious software. Apart from that, all of them are focused on the Android platform or IoT platform for malicious software detection, classification, and defense. However, there are no papers related to countermeasures against ELF malicious software on the Linux_x86/ARM platform.

1.1 Background Overview

3. Currently, the existing work on countermeasures against PE malicious samples cannot generate executable binary files.

The generation of adversarial malicious software consists of two types:

The first type is to extract features from binary malicious software and then use adversarial sample techniques to add perturbations to the data in the feature domain, so as to deceive the malicious software detection system. However, the perturbations made by the attacker are restricted to the feature domain and do not generate adversarial executable binary files.

The second type is an attack on malware detection systems based on image classification. Such type of detection system first converts binary malware into grayscale images, and then classifies the malware using image classification techniques. Attackers use adversarial sample techniques to add perturbations to grayscale images, so as to deceive image-based malware detection systems. No executable binary files are generated.

Neither of the above two methods generates adversarial executable binary files.

1.1 Background Overview

4. Currently, there is no work on AI-based generation of malicious ELF samples.

The existing PE Anti-AntiVirus ("Virus AV") techniques are based on traditional Virus AV techniques such as encryption, obfuscation, and packing. Nonetheless, there is still no work on using AI technology to generate malicious ELF samples. Generation of adversarial malicious ELF samples using AI technology is a blank area and a challenging task.

5. Development of Automated Systems

In general, traditional Virus AV techniques for PE samples involve manual modification and testing by engineers, which is a tedious and inefficient process. Currently, there are no papers or related works that can automatically generate adversarial malicious samples in PE format. The current research on ELF is focused on defense, with no work done on adversarial attacks. It is a challenging task to generate adversarial ELF samples in an automated manner.

1.2 Technical Application Scenarios

The project establishes a sample foundation for the blue team and threat intelligence departments. It can be adopted to generate undetectable samples in security attack and defense scenarios based on this technological achievement



One
The project can be used to support AV for improving its detection capabilities. By testing on AV, it can discover the flaws in AV itself, thereby enhancing the robustness of AV and solving the security issues of AV models to a certain extent



PART TWO

TECHNICAL
OBJECTIVES

2.1 Applicable Scenarios

In the environment of UOS or Ubuntu Kylin domestic operating system, under the constraints of specific compilation switches, commercial security software, defense methods, and relative escape rate indicators, AI technology is used to realize a prototype system for automated generation of executable ELF adversarial malicious samples with a web interface

Applicable Scenarios

The input is an ELF sample, and the output is an executable malicious ELF sample that meets certain escape rate indicators

2.2 Technical Challenges

1

The first research on adversarial malicious samples in the ELF format on domestic platforms

Most of the defense work against malicious ELF software only targets ARM architecture platforms such as IoT and Android. Nonetheless, there is zero research on malicious ELF software on domestic platforms (domestic operating systems UOS/Kylin) with Linux x86/ARM architecture

2

This solution can generate adversarial malicious samples targeting the ELF format

The existing research on malicious samples is primarily focused on the PE format, yet there is very little work on ELF format samples. Moreover, the few existing works are focused on defense against ELF malicious samples, and there is still insufficient research on adversarial attacks against ELF format samples. This solution can generate adversarial malicious samples in ELF format for the Linux platform.

3

Capacity to generate executable binary files

Most of the current work on PE format malicious samples only pertains to perturbations in the feature and image domains, without generating executable binary files. The generation of executable malicious samples in ELF format remains an open area of research. However, this proposed AI-based automated generation system can generate executable binary files in ELF format.

4

AI-Based Automated Generation System for Adversarial Malicious ELF Samples

The existing Virus AV techniques for PE malware require engineers to manually modify and test binary files or malicious source code, and then modify and test again. Such process is very cumbersome and inefficient. Beyond that, there is a blank space for manual Virus AV techniques for ELF malware. The proposed AI-based automated generation system can efficiently and automatically generate adversarial malicious ELF samples.

AI-Based Adversarial Malicious Sample Generation Scheme

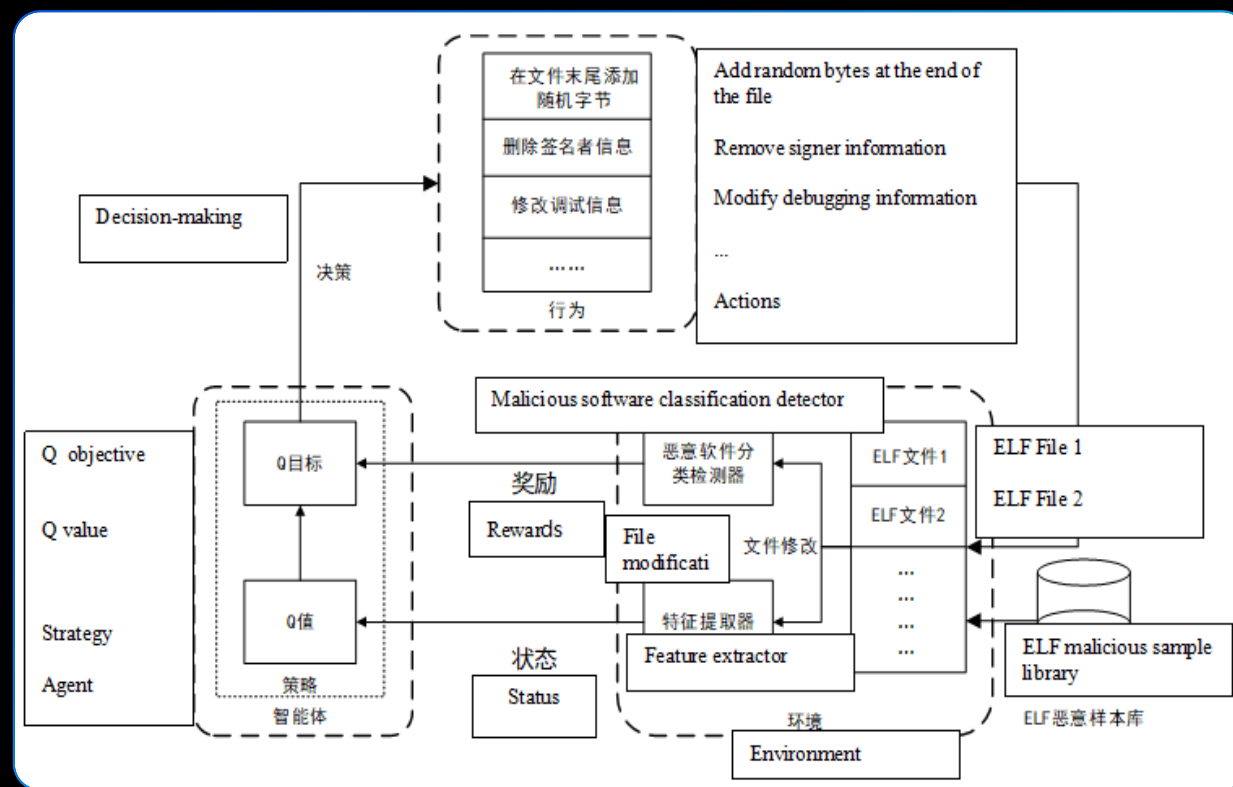


Figure 1: Flowchart of Adversarial Malicious Sample Generation Based on Reinforcement Learning

AI-Based Adversarial Malicious Sample Generation Scheme

(1) Definition

Reinforcement learning is a process where an intelligent agent continuously interacts with the environment to strengthen its decision-making abilities. First, the environment (ENV) provides an observation (also called state) to the agent based on AI. After receiving the observation from the environment, the agent takes an action. Then, the environment reacts to the action by offering a reward and a new observation (state). The intelligent agent updates its policy based on the rewards given by the environment, and the goal of reinforcement learning is to obtain the optimal strategy.



AI-Based Adversarial Malicious Sample Generation Scheme

(2) Model structure of this solution

Environment

The environment consists of two modules: a malicious software classification detector and a malicious sample feature extractor.

The malicious software classification detector refers to the target commercial security software that needs to be bypassed. The feature extraction module extracts features from modified malicious samples using the LIEF tool library. The more accurate the extracted features are, the more advantageous it is for the intelligent agent to make the next decision based on the observed values (states).

Agent

The agent makes decisions on the next modification based on the rewards and states provided by the environment through the policy. The policy used here is Q-Learning, which establishes a Q-table via a reward and punishment mechanism to determine how to generate the next decision.

AI-Based Adversarial Malicious Sample Generation Scheme

(2) Model structure of this solution

Action

The action module includes multiple methods for modifying malicious ELF files, such as adding random bytes to the end of the malicious sample, creating a new file section, deleting signer information, and modifying debug information.

Based on the common detection rules used by malicious software static analysis tools, the methods in the action module are determined.

The behavior module will modify the malicious ELF software in the environment (env) based on the next decision made by the agent, and generate modified malicious ELF samples by combining multiple behaviors.

AI-Based Adversarial Malicious Sample Generation Scheme

(3) Generation Process

01

Sample Extraction

The ELF malicious sample S is extracted from the ELF malicious sample library and its corresponding label is read

02

The environment module analyzes the extracted ELF malicious sample.

The malware detector analyzes S to determine whether it is malicious software and provides a reward to the agent.

The feature extractor extracts features from S (e.g. signature information) and provides these states to the agent

03

The intelligent agent analyzes S based on the environment and makes decisions using the DQN algorithm. It modifies S through methods in the behavior module to obtain S'

04

Obtain S' as S and repeat step 2 until the malicious software detector in the environment module determines it as benign

AI-Based Adversarial Malicious Sample Generation Scheme

Furthermore, the method of generating adversarial malicious ELF samples based on GAN is explored, with the following approach:

1. The generator utilizes randomly generated binary bytes (random noise) and adds them to malicious samples extracted from the ELF malicious sample library to generate modified malicious samples.

2. The discriminator compares benign samples extracted from the benign ELF sample library with modified malicious samples, makes judgments, provides feedback to the generator, and participates in the next generation process.

Through continuous iteration of these two steps, the discriminator's ability to distinguish gets stronger, and the adversarial malicious samples generated by the generator become closer to benign samples, until the discriminator classifies the generated adversarial malicious samples as benign, and the iteration ends

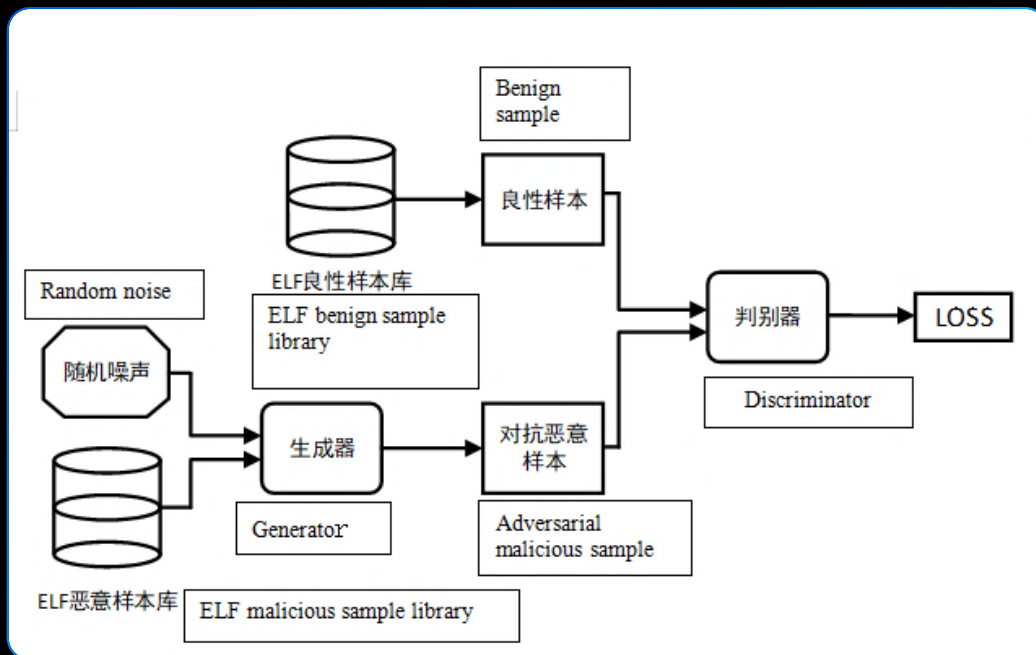


Figure 2. Flowchart of generating adversarial malicious samples based on GAN



PART THREE

TECHNICAL
SOLUTIONS

3.1 Summary of the Solution

In the environment of UOS or Ubuntu Kylin domestic operating system, under the constraints of specific compilation switches, commercial security software, defense methods, and relative escape rate indicators, AI technology is used to realize a prototype system for automated generation of executable ELF adversarial malicious samples with a web interface.

Summary of the Solution

This system mainly realizes a reinforcement learning-based ELF Virus AV system, and is implemented under the Galaxy Kylin system. Based on Python 3.8, it combines with clamAV and Go language. The important Python libraries used include gym for building reinforcement learning environments, lief for parsing ELF files, and numpy for data processing. In addition, the system can not only read important information of ELF files in 64-bit and 32-bit domestic system environments but also provide corresponding Virus AV behavior tables for ELF structures to achieve virus Virus AV. In addition, the project provides interaction with domestic antivirus software clamAV and can train reinforcement learning agents for the sample set and store the training result models for future use.

3.2 Construction of Malicious ELF Sample Database

This study collects and gathers malicious ELF programs from the internet, performs preprocessing tasks such as running and debugging, adds classification labels, and creates a sample library that can be used as a testing benchmark. The integrity and maliciousness of the samples are verified.

725 X86 ELF viruses

303 ARM ELF viruses

21 cross-platform compiled virus source codes

```
cuckoo@ubuntu:/cm$ readelf -h 0a1a8ca1ce27a04bf9618fe0f6bc94e6
ELF 头:
Magic:      7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
类别:      ELF64
数据:      2 补码, 小端序 (little endian)
版本:      1 (current)
OS/ABI:    UNIX - System V
ABI 版本:  0
类型:      EXEC (可执行文件)
系统架构:  Advanced Micro Devices X86-64
版本:      0x1
入口点地址: 0x1078b8
程序头起点: 64 (bytes into file)
Start of section headers: 0 (bytes into file)
标志:      0x0
本头的大小: 64 (字节)
程序头大小: 56 (字节)
Number of program headers: 3
节头大小:  64 (字节)
节头数量:  0
字符串表索引节头: 0
cuckoo@ubuntu:/cm$
```

Figure 3.Obtaining virus sample-related information using readelf

Security vendors' analysis	Threat categories	Family labels
Ad-Aware	Trojan.Linux.Generic.166385	Linux/Ransom.139348
ALYac	Trojan.Ransom.Linux.Gen	Trojan(Ransom)/Linux.Crypnux.a
Arcabit	Trojan.Linux.Generic.D289F1	ELF.Filecoder-B [Cryp]
AVG	ELF.Filecoder-B [Cryp]	LINUX/Cryptor.saa.2
BitDefender	Trojan.Linux.Generic.166385	Malware@#15771e24zj9g8
Cynet	Malicious (score: 99)	Linux.Encoder.1
Emisoft	Trojan.Linux.Generic.166385 (B)	Trojan.Linux.Generic.166385
ESET-NOD32	Linux/Filecoder.A	ELF/Filecoder.Altr
GData	Trojan.Linux.Generic.166385	Detected
Ikarus	Trojan.Ransom.Linux.Encoder.One	Trojan.Linux.ag
Kaspersky	Trojan.Ransom.Linux.Cryptor.b	Trojan.Linux.Cryptor.jlc
MAX	Malware (ai Score=100)	Linux/Ransom
McAfee-GW-Edition	Linux/Ransom	Ransom.Linux/Crypnux
NANO-Antivirus	Trojan.ElR94.Ransom.ebdddv	Linux/Ransom.A
Rising	Ransom.Encoder/Linux1.A3F7 (CLOUD)	Suspicious.Linux.Save.a
Sophos	Linux/Ransm-C	Unix.LinuxEncoder
Tencent	Linux.Trojan.Cryptor.Agow	Trojan.Linux.Generic.166385
TrendMicro	ELF.CRYPTOR.A	ELF.CRYPTOR.A

Figure 4.Screenshot of VirusTotal Detection Results

Classification of ELF virus samples & detection of maliciousness in ELF virus samples: The VirusTotal tool is used to obtain static information (category, version, entry address, etc.) and individually test the virus samples.

3.2 Construction of Malicious ELF Sample Database

A local detection sandbox (based on Tencent HaBo) has been set up. Through the use of HaBo, virus samples are tested one by one to obtain dynamic information about the samples (processes, files, networks).

Process	
Behaviour:	Execute a file
Detail info:	execve: /home/nuaa/haBo/virus/VirusShare_1e19b857a5f5a9680555fa9623a88e99 execve:
Behaviour:	Process exit
Detail info:	procexit status=0 ret=0 sig=0 core=0 procexit status=11 ret=0 sig=11(SIGSEGV) core=0

Figure 5. HaBo Detection of Process-related Information

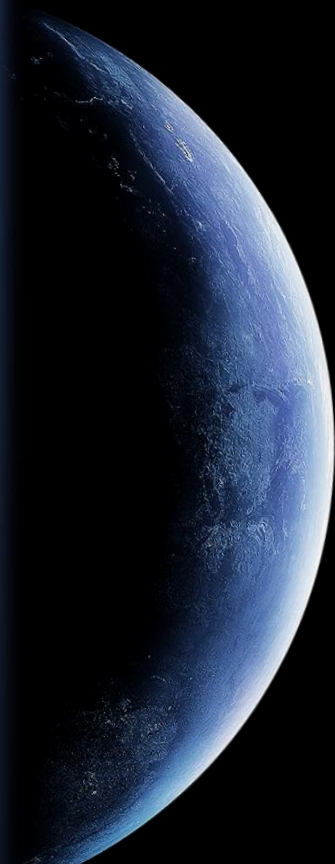
File	
Behaviour:	File read
Detail info:	read: path=/proc/16017/stat, size=0 read: path=/proc/747/cmdline, size=0 read: path=/proc/53260/stat, size=175 read: path=/proc/77/stat, size=0 read: path=/proc/53260/cmdline, size=109 read: path=/proc/3172/cmdline, size=0 read: path=/proc/9/stat, size=0 read: path=/proc/19/cmdline, size=0 read: path=/proc/53524/stat, size=0 read: path=/etc/host.conf, size=0 read: path=/proc/596/stat, size=0 read: path=/proc/2369/stat, size=172 read: path=/proc/489/stat, size=0 read: path=/proc/113/cmdline, size=0 read: path=/proc/53664/stat, size=309

Figure 6. HaBo Detection of File-related Information

Network	
Behaviour:	SSL certification
Detail info:	*.uniontech.com is issued by GeoTrust CN RSA CA G1 GeoTrust CN RSA CA G1 is issued by DigiCert Global Root CA
Behaviour:	connect
Detail info:	connect: 192.168.222.132:53188->101.37.43.176:1025 connect: 192.168.222.132:47382->180.101.50.188:0 connect: 192.168.222.132:60716->192.168.222.2:53 connect: 0.0.0.0:0->0.0.0.0:0 connect: 192.168.222.132:42581->192.168.222.2:53 connect: 192.168.222.132:42025->180.101.50.242:1025 connect: 192.168.222.132:44157->192.168.222.2:53 connect: 192.168.222.132:38533->192.168.222.2:53 connect: 192.168.222.132:56650->192.168.222.2:53 connect: 192.168.222.132:34112->101.37.43.176:1025 connect: 192.168.222.132:42766->192.168.222.2:53 connect: 192.168.222.132:60005->180.101.50.242:0 connect: 0->ffff9e23c7acf400 /var/run/nscd/socket
Behaviour:	socket
Detail info:	socket: domain=16 type=524291 proto=0 socket: domain=10 type=3 proto=58 socket: domain=10 type=2 proto=58 socket: domain=2 type=2 proto=1 socket: domain=2 type=526338 proto=0 socket: domain=16 type=526339 proto=0 socket: domain=2 type=3 proto=1 socket: domain=2 type=2 proto=0 socket: domain=1 type=526337 proto=0 socket: domain=2 type=524290 proto=0

Figure 7. HaBo Detection of Network-related Information

3.3 AI-Based Generation Technology for Adversarial Malicious ELF Samples



This project consists of four modules: reinforcement learning agent construction, local clamAV Virus AV tool construction, reinforcement learning environment construction, and Virus AV behavior table construction. To be specific, the reinforcement learning agent construction module mainly focuses on agent training and actual operation code. Based on this code, the necessary function interfaces for using the agent are encapsulated to improve the usability of the entire reinforcement learning Virus AV system. The local clamAV Virus AV tool construction module mainly implements code encapsulation for the scanning tool clamAV and the extraction of ELF file information. Particularly, this module designs nine feature classes for extraction of ELF file information based on the characteristics of ELF file structure, and integrates them into a feature vector extraction class for easy extraction of ELF file feature vectors. The reinforcement learning environment construction module sets up a usable reinforcement learning environment for clamAV according to the OpenAI gym environment construction standards, and enables the reinforcement learning agent to interact with clamAV in real-time in the environment, so as to improve the efficiency of the Virus AV process. The Virus AV behavior table construction module implements Virus AV operations for ELF files based on the current theoretical research. These Virus AV operations include modification of ELF file header information, addition of redundant information to ELF files, addition of useless section information to ELF files, modification of ELF file symbol information and dynamic library information, and implementation of ELF file encapsulation using the Go language. This Virus AV behavior table is encapsulated and provided for use by the reinforcement learning agent.

3.3 AI-Based Generation Technology for Adversarial Malicious ELF Samples

Figure 8 shows the basic structure diagram of the scheme. Based on the architecture of reinforcement learning, the entire system is divided into multiple modules according to the requirements of the Virus AV system.

This diagram briefly illustrates the design structure of the scheme: The virus detection tool encapsulates the sample as a reinforcement learning environment. The virus detection tool extracts the sample for detection and submits the detection result and the file feature vector, which is the state, to the agent. The agent selects actions based on the learned policy and modifies the sample. Beyond that, the system obtains the newly generated sample and submits it to the virus detection tool for detection. The virus detection tool returns the detection result and the state to the agent again. This process continues until successful Virus AV or failure. Moreover, the system defines that if the Virus AV operation does not achieve the Virus AV effect after a certain number of iterations, it is considered a failure.

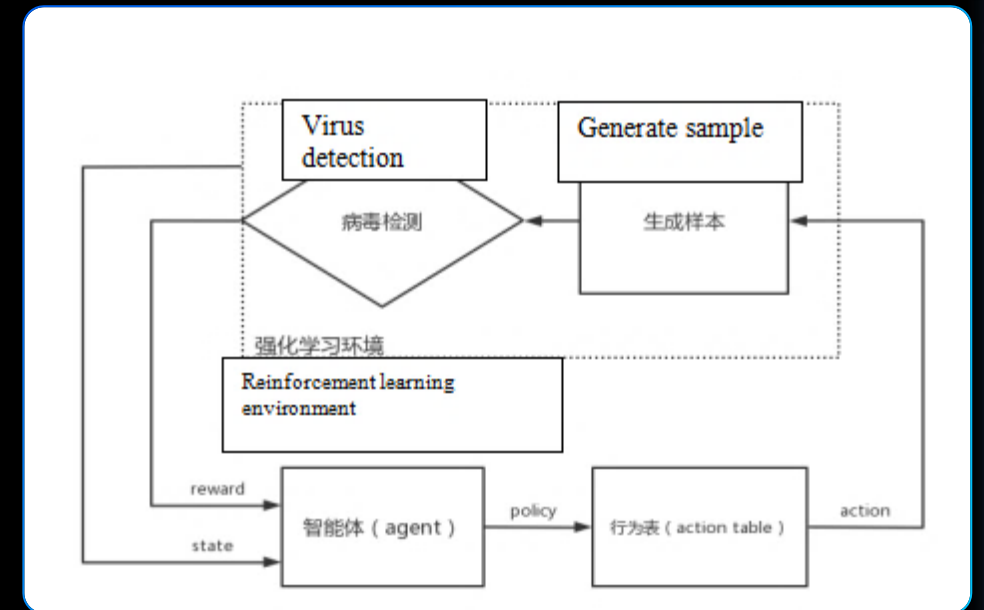


Figure 8.Reinforcement Learning ELF Virus AV Architecture

3.3 AI-Based Generation Technology for Adversarial Malicious ELF Samples

Overview of Reinforcement Learning-Based Virus AV Methods

- ✦ Modifying actions targeting PE to target ELF based on the structure of ELF files
- ✦ Building a local ClamAV antivirus engine and interacting with reinforcement learning
- ✦ Studying the working principle of feature vector extraction in antivirus software
- ✦ Exploring and researching the extraction of virus feature vectors
- ✦ Developing and implementing the extraction of virus feature vectors, especially for complex viruses, extracting complex feature vectors, and debugging and running, writing and implementing the extraction of feature information for symbols, dynamic linking, libraries, import, and export
- ✦ Development and Expansion of Action Table

Effect

The modified and ported existing actions are only effective against simple viruses. The developed actions based on virus feature vectors are also effective against complex viruses and can greatly improve the success rate of Virus AV.

3.4 Implementation Plan

3.4.1 Construction of Reinforcement Learning Agents

This part is relatively easy in the project. The PPO algorithm based on Python 3.8+stable_baselines3 is used as the core of the agent. An API class is defined to encapsulate the core of the agent, which mainly includes the code for constructing the agent training part and the agent running part. In addition, there is also code for Virus AV iteration and reinforcement learning environment initialization, which is encapsulated as function interfaces for the future use. The agent training part is implemented based on the learn method provided by the aforementioned PPO algorithm module and will save the trained agent model upon the completion of training. Thus, there is no need to elaborate on the corresponding code. The agent running function is the main function interface of the agent building part, and its algorithm pseudo code can be shown as follows.

```
Function evade_action():
Begin
  Var reward:=0,done:=False
  For iteration:=first_sample_file to last_sample_file:
    Var ob:=env.reset()
    While True do
      Var action,_states:=predict(ob,reward,done)
      ob, reward, done, history:=step(action)
    If done==True and reward>=10 Then
      Write after evade file as benign
    Else
      Write after evade file as malicious
  End
```

```
End
  Write after evade file as malicious
Else
  Write after evade file as benign
```

3.4 Implementation Plan

3.4.1 Construction of Reinforcement Learning Agents

The above pseudocode explanation: The main part of running the agent is a two-layer loop, in which the outer loop is responsible for iterating through all the samples in the sample set. The inner loop is an infinite loop that repeatedly performs Virus AV operations on the sample iterated by the outer loop until Virus AV is successful or too many iterations lead to Virus AV failure. Here, the maximum number of iterations is set to five. If the maximum number of iterations is not set, the agent may continue using a certain anti-detection operation. Although it may still achieve Virus AV in the end, the behavior of constantly using a certain Virus AV operation may become a new feature of the virus. Therefore, the research group recommends that the maximum number of iterations for a single sample shall not exceed five times.

The basic flowchart of the reinforcement learning Virus AV system encapsulation is presented in Figure 9:

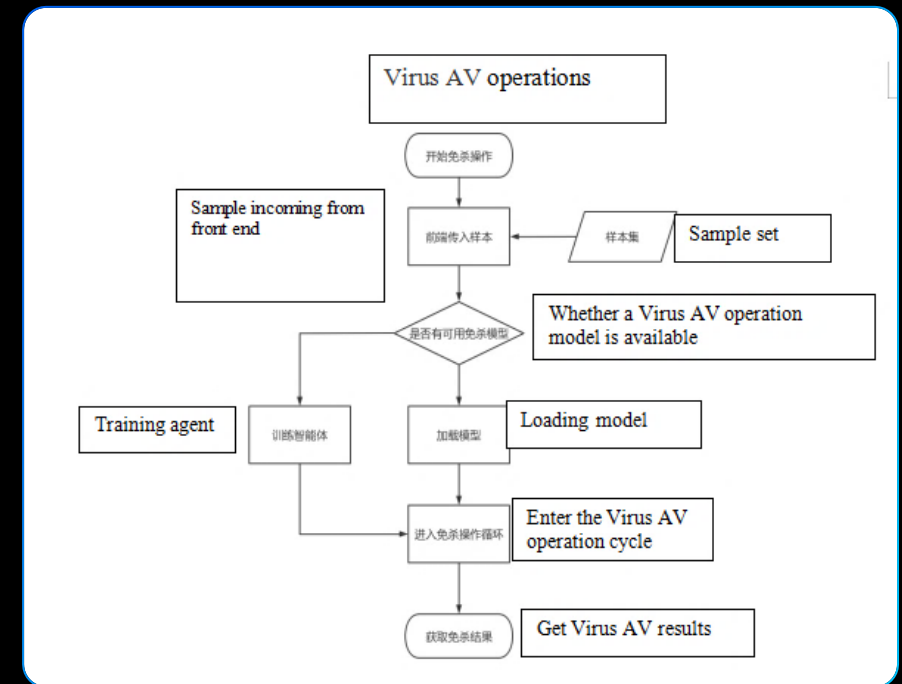


Figure 9. Flowchart of the reinforcement learning anti-detection system operation

3.4 Implementation Plan

3.4.1 Construction of Reinforcement Learning Agents

Figure 9 shows the basic operation flow of the reinforcement learning Virus AV system after function encapsulation. In addition, the system provides two modes: retraining the intelligent agent for the sample and directly loading the existing intelligent agent model. When the front end calls the system, it should first pass the sample set to the system, and then select the corresponding intelligent agent based on whether the system needs or has an available Virus AV model. Besides, the system enters the actual Virus AV operation in the above-mentioned reinforcement learning Virus AV operation function. Finally, the system obtains the Virus AV result and returns the Virus AV result history and the Virus AV result storage path to the front end.

3.4 Implementation Plan

3.4.2 Construction of a Local ClamAV Bypass Tool

Currently, there are many well-performing virus scanning tools in the domestic environment, but most of them fail to provide function interfaces. ClamAV is a commonly used free and open-source virus scanning tool for Linux-like systems. With strong virus detection capabilities, this tool provides a user-friendly function interface based on the Python language, so that it is easy to read detection results and other features. Therefore, in the project, ClamAV is selected as the main scanning tool to interact with the reinforcement learning agent. Other domestic scanning tools will also be introduced in the experiment to scan and compare the results of the samples that have been bypassed by ClamAV.

The system rewards the agent with different scores based on the detection results of ClamAV.

Detection results	Reward scores
Benign	10
Malicious	0

3.4 Implementation Plan

3.4.2 Construction of a Local ClamAV Bypass Tool

Firstly, a brief explanation is needed on the basic usage of the function interface provided by ClamAV in Python. In Python, the `pyclamd` library is used to implement code-level calls to the local ClamAV in the Python language. The main function interfaces are as follows:

- (1) `cd = pyclamd.ClamdAgnostic()`, this function interface is adopted to instantiate a Clamd control object, and subsequent encapsulation mainly revolves around this object.
- (2) `cd.ping()`, this function interface tests the connectivity of Clamd.
- (3) `cd.scan_stream()`, this function is utilized to detect byte streams. In the solution, this function is mainly selected as the method for scanning sample bytes, because it requires modification of the samples at the binary level and submission of the modified results to the virus detection tool.

3.4 Implementation Plan

3.4.2 Construction of a Local ClamAV Bypass Tool

It is noteworthy that the malicious results in the virus detection results provided by pyclamd are a Python dictionary, and the values in the dictionary key-value pairs are a Python tuple, which stores detailed information about the virus detection results. The key part is the string 'stream', whereas the results that show non-malicious behavior are just a None value. Therefore, it is necessary to further encapsulate the scan_stream of pyclamd so as to unify the format of the detection results.

As the main content of this part, the Clamav class further encapsulates the function interface provided by pyclamd and provides the observe_env function as the main interaction interface. observe_env has three main functions:

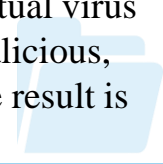
01

calling scan_stream to obtain the detection results;



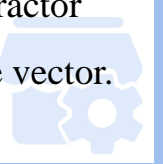
02

unifying the detection results into the structure {'stream': (scan result)}, where the scan result is the actual virus detected when the result is malicious, and the string 'None' when the result is non-malicious;



03

calling the feature extraction function provided by the ELFFeatureExtractor class and passing out the feature vector.



3.4 Implementation Plan

3.4.2 Construction of a Local ClamAV Bypass Tool

ELF file feature vectors are extracted and encapsulated in the project using the ELFFeatureExtractor class, which includes nine different feature classes: ByteHistogram, ByteEntropyHistogram, StringExtractor, GeneralFileInfo, HeaderFileInfo, SectionInfo, SymbolsInfo, ImportsInfo, and ExportsInfo. The principles of these nine feature classes are briefly explained below.

The ByteHistogram class

The ByteHistogram class mainly converts the binary content of the ELF file into an array of eight-bit unsigned integers. Then, it counts the occurrences of each number in the array and organizes them into a new array called the histogram array. Finally, all elements in the histogram array are divided by the sum of all elements in the original array to acquire the byte histogram information represented by this class.

The ByteEntropyHistogram class

The ByteEntropyHistogram class is similar to the ByteHistogram class and mainly obtains the byte entropy histogram information of the file. This class first constructs a window of 2048 bits in length for extracting file information, and specifies that the window will move with a step size of 1024 bits upon extraction of information. Similar to the ByteHistogram class, the ByteEntropyHistogram class converts the binary content of the ELF file into an array of eight-bit unsigned integers and converts the array result into a histogram array and divides it by the window size, and then calculates it via the formula

$$H = \text{sum}(-p[idx] \times \log_2(p[idx])) \quad (1)$$

Operations are performed on each element of the statistical array and the sum of the operation results is stored as the main feature information for this part. In the formula: H represents the sum result, sum represents the summation function, p represents the array obtained by dividing the statistical array by the window size, log₂ represents the logarithm function with base 2, and idx represents the array index used for accessing each element.

3.4 Implementation Plan

3.4.2 Construction of a Local ClamAV Bypass Tool

The StringExtractor class

The StringExtractor class is used to extract readable strings, file paths, URLs, and other string information from the binary content of ELF files. This class adopts the re library in Python to standardize the format of the feature string information to be extracted and uses the findall function of the re library to extract all special string information from the binary file. Given that the feature vector describes the file in numerical form, this class also calculates string entropy, string length, and other information after extraction of strings from ELF files to construct the feature vector of this class. The formula for calculating string entropy is similar to formula

The GeneralFileInfo class

The GeneralFileInfo class uses the ELF object provided by the lief library in Python to obtain important global information from ELF files, including the size of the ELF file, virtual size of the ELF file, string length, export function information, import function information, redundant information at the end of the ELF file, whether the ELF file has redundant information, import library information, and symbol information. Finally, the information is processed via data processing libraries such as numpy in Python to convert it into a feature vector. The design of this class is mainly based on the design of extracting the main information from PE files in the paper, which provides the main global information about ELF files for reinforcement learning agents.

The HeaderFileInfo class

The HeaderFileInfo class is also implemented based on the ELF object provided by the Python lief library and related research. This class mainly describes the main information of the ELF header in the ELF file and constructs a brief model of the ELF header. The main information extracted by this class encompasses the machine architecture on which the file runs (such as ARM, X86, and X64), file type (shared library file, executable file, etc.), ELF header size, number of ELF sections, number of ELF segments, ELF file entry point, etc. The class extracts various types of information, in which some are numbers and some are strings. The class unifies multiple types of information by calculating fixed-length hash values and uses them as the feature vector information of the class.

3.4 Implementation Plan

3.4.2 Construction of a Local ClamAV Bypass Tool

The SectionInfo class

The SectionInfo class mainly extracts information about the sections of an ELF file. This class extracts the name, size, entropy value, and permissions of each section. Due to the diverse types of information in this class, the approach of calculating a fixed-length hash value and calculating the entropy value is used to convert string and other types of information into numbers as the feature vector information of this class.

The ImportsInfo class

The ImportsInfo class extracts information as part of the SymbolsInfo information. This class extracts function symbols and variable symbols from the imported symbols in the ELF file, together with the imported libraries in the ELF file as the original information. Likewise, by calculating a fixed-length hash value as the feature vector of this class, this class converts various import symbol information into fixed-length numerical values.

The SymbolsInfo class

The SymbolsInfo class, which is mainly designed based on the research, extracts symbol information from ELF files. Beyond that, this class divides symbols in ELF files into function symbols, variable symbols, static symbols, dynamic symbols, import symbols, and export symbols, and counts the number of each type of symbol. The counting results are used as the feature vector information of this class.

ExportsInfo class

ExportsInfo class is similar to ImportsInfo class. It mainly extracts various kinds of information about exported symbols in ELF files and converts them into feature vectors of this class through calculation of fixed-length hash values.

3.4 Implementation Plan

3.4.3 Construction of Reinforcement Learning Environment

This section mainly builds the reinforcement learning environment based on the gym framework. According to the requirements of the gym reinforcement learning framework, the self-built reinforcement learning framework should be encapsulated through a class and must have the step function, `_take_action` function, and reset function. To be specific, the step function is mainly responsible for the execution steps required for a single iteration of the agent and the return of the results after iteration. The `_take_action` function is primarily adopted to select the action taken during a single iteration, and the reset function is mainly applied to reset the reinforcement learning environment, including resetting relevant parameters and resetting samples. The reinforcement learning environment of ClamAV is encapsulated as a `ClamavEnv` class and provides `_take_action`, `reset`, and `step` functions as interaction interfaces. The implementation ideas of the reset function and step function are introduced as follows:

The pseudocode for the reset function in the `ClamavEnv` class is as follows:

```
Function reset()
Begin
  Var Turns:=0
  While True do
    If random_choice==True then
      File:=random(files)
    Else
      File:=files[idx%file count]
      idx+=1

    answer, observation space:=scan file(file)
    If answer is benign then
      continue

    break
  End
```


3.4 Implementation Plan

3.4.3 Construction of Reinforcement Learning Environment

The above pseudocode briefly explains the working process of reset, which is to reset the main variables in the environment such as the iteration count. Then, it enters a loop to select a sample from the sample set that is detected as malicious for the next iteration and exits the loop. If the chosen sample is detected as non-malicious, the loop will continue.

Given that the non-malicious samples are directly skipped, in the actual usage, only the samples detected as malicious by ClamAV in the input sample set can participate in the iteration. In order to ensure that the frontend obtains undetected samples corresponding to its input sample set, this function will also incorporate the first sample detected as non-malicious in the iteration, but no operation is performed on it.

The pseudocode for the step function is as follows:

```
Function step(action_ix):
Begin
  _take_action(action_ix)
  answer,observation_space:=observe_env(bytez)
  result=predict(bytez)
  If result==True then
    Reward:=10
    Done:=True
  Elif turns>=max_turns then
    Reward:=-10
    Done:=True
  Else
    Reward:=-10
    Done:=False
  Return observation_space,Reward,Done,answer
End
```

```
End
Return observation_space,Reward,Done,answer
Done:=False
Reward:=-10
Else
```

3.4 Implementation Plan

3.4.3 Construction of Reinforcement Learning Environment

According to the above pseudocode, the step function takes the iteration number `action_ix` as input and selects and executes the corresponding behavior from the Virus AV behavior table. The reward value obtained in every time of iteration is determined by the detection result obtained after execution of the behavior and the number of iterations. The feature vector, reward value, termination flag, and detection result are returned as a quadruple. The `observation_space` in the quadruple is the feature vector extracted by the reinforcement learning environment, `Reward` signifies the reward value returned in the current iteration, `Done` signifies the flag indicating whether the current iteration is finished, and `answer` is mainly used for debugging the code according to the description of the gym library, which exerts no impact on the system operation.

3.4 Implementation Plan

3.4.4 Construction of Virus AV Behavior Table

This section mainly focuses on constructing Virus AV behaviors that modify the ELF structure without completely destroying the ELF file structure, thereby allowing it to run normally. Most of these Virus AV behaviors involve addition of meaningless content to the ELF file to confuse the signature. Below is an explanation of Virus AV behaviors.

The first behavior is adding redundant data to the end of the ELF file. This behavior disrupts the file checksum, and makes it impossible to use checksum-based signatures for virus detection. Based on this method, four behaviors have been implemented, in which one of them will be selected as a representative to explain the implementation approach.

3.4 Implementation Plan

3.4.4 Construction of Virus AV Behavior Table

`append_benign_data_overlay` is one of the representative behaviors among the four behaviors of adding redundant data to the end of a file. This behavior not only selects a non-malicious file from a clearly labeled non-malicious file sample set but also uses the parsing function provided by the `lief` library to parse the file binary into an ELF file object. Then, it extracts the `.text` section content of the non-malicious file as redundant information and adds it to the end of the ELF file binary being processed. The other three behaviors of adding redundant data to the end of the file are similar to this behavior. `pad_overlay` adds a series of random numbers, `append_benign_binary_overlay` directly adds the content of the non-malicious file to the end of the file, and `add_strings_to_overlay` adds non-malicious file strings to the end of the file.

3.4 Implementation Plan

3.4.4 Construction of Virus AV Behavior Table

Next is the operation of modifying ELF file sections. There are four types of such operations: adding useless sections, adding string obfuscation sections, adding obfuscation data to section gaps, and modifying section names. Below is an explanation of these four operations.

The function `add_section_benign_data` selects a non-malicious file from a non-malicious file sample set and selects a section from it to add to the ELF file being processed. The pseudocode can be described as follows:

```
Function add_section_benign_data():  
Begin  
    benign_file:=random(benign_files)  
    benign_sections:=benign_file.sections  
    benign_section:=random(benign_sections)  
    If current file has section then  
        current_file.add(benign_section)  
End
```

3.4 Implementation Plan

3.4.4 Construction of Virus AV Behavior Table

This function adds useless sections to the ELF file, and modifies the section header table of the ELF file, the content of the .shstrtab section, and the mapping between other sections and the section names in the .shstrtab section. This can, to some extent, affect the offset of certain feature codes in the sections, making it difficult to accurately locate them. Given that the ELF samples targeted by this system tend to be ELF files in the runtime view, the section header table may not exist, making it difficult for the system to accurately locate a section in the system. Therefore, it is necessary to check whether the current file has section headers before addition of a section.

3.4 Implementation Plan

3.4.4 Construction of Virus AV Behavior Table

The `add_section_strings` function not only selects non-malicious strings extracted from non-malicious samples but also chooses one random section name that is not existent in the current processed sample file to form a new section with the non-malicious characters. Then, this section is added to the processed ELF file. The pseudocode for this function can be described as follows:

```
Function add_section_strings():
Begin
    good_strings:=random(good_string_sample)
    section_name:=random(available_section_names)
    new_section:=create(section_name, good_strings)
    If current file has section then
        current_file.add(new_section)
End
```


This function mainly influences certain feature code offsets by adding file sections to the file.

The `add_bytes_to_section_cave` function reads all sections in the processed ELF file that have a length greater than or equal to 128 bytes and searches for random gaps of the corresponding length. Then, it generates random obfuscation data of the same length and replaces the original gaps, thereby adding obfuscation data to the gaps. The pseudocode for the file gap search involved in this operation can be described as follows:

```
Function _search_cave(section_content, section_offset, section_name)
Begin
    For byte:=first_byte to last_byte:
        check:=False
        If byte is b'\x00' then
            count+=1
        Else
            check:=True
        If byte is last_byte then
            check:=True
        If check is True and count>=128 then
            found_caves.append([cave_start,cave_end,cave_size,section_name])
            count:=0
    Return found_caves
End
```


3.4 Implementation Plan

3.4.4 Construction of Virus AV Behavior Table



After calling the function to search for a cave, `add_bytes_to_section_cave` will determine whether there is an available gap in the current operating file. If the gap is found, it will randomly select one gap and generate random data of the corresponding length to replace it. Given that this operation deals with blank data in file sections, it does not directly damage the ELF file being operated on.

The `rename_section` operation is relatively simple. It randomly selects a file name from the sections of the ELF file being operated on, and meanwhile randomly selects a file name from the file names that the file does not have to replace the selected file name in the ELF file.

3.4 Implementation Plan

3.4.4 Construction of Virus AV Behavior Table

Next are the `add_imports` operation for adding file import information and the `add_library_` operation for adding file import libraries. These two behaviors have some similarities. In `add_imports`, this operation selects the dynamic symbols used in non-malicious files and adds them to the ELF file being processed. Likewise, `add_library_` selects dynamic libraries from non-malicious files and adds them to the ELF file being processed.

In addition, there is also a behavior called `modify_machine_type` for modifying the ELF file header. This behavior mainly uses the interfaces provided by the `lief` library to retrieve and modify the architecture section of the ELF file header.

Finally, notably, `go_bindata` is an operation that wraps the ELF binary file using the `bindata` implementation in the Go language. This operation encapsulates the ELF binary file to be wrapped as the content of the data section in a new ELF file. The new ELF file is written and compiled in the Go language, and its main function is to load and run the encapsulated ELF file in memory.

ELF binary files generated by the Go language always link the runtime library during static linking. This library consists of garbage collection, thread scheduling, and other functions. Thus, go language files have large numbers of functions, and general Go language functions tend to involve many other functions. It is difficult to distinguish the maliciousness of many functions in Go language files. Therefore, the detection efficiency of mainstream virus detection tools for ELF files generated by the Go language is often average. Based on this, this feature can be used to achieve Virus AV of antivirus detection.

3.4 Implementation Plan

3.4.4 Construction of Virus AV Behavior Table

Additional Function: Anti-detection method based on UPX

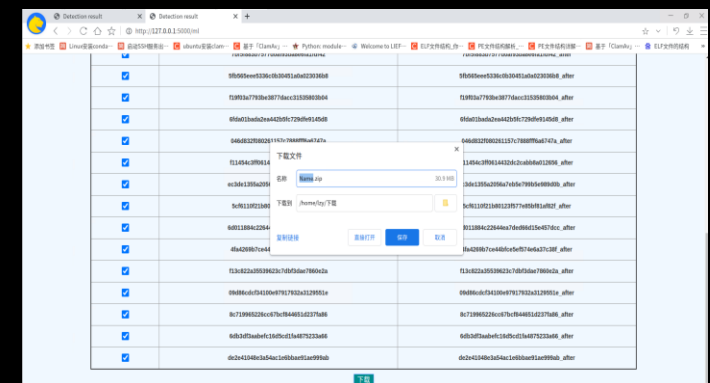
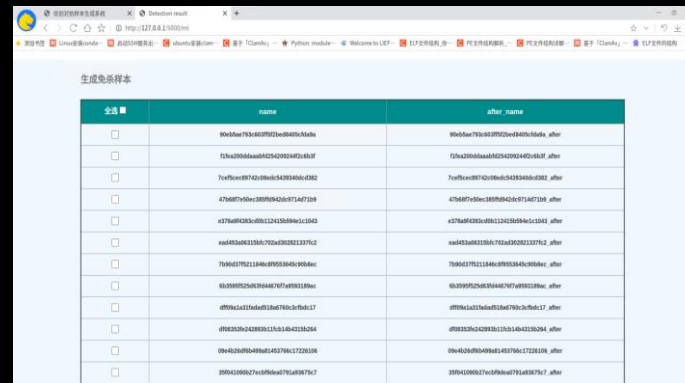
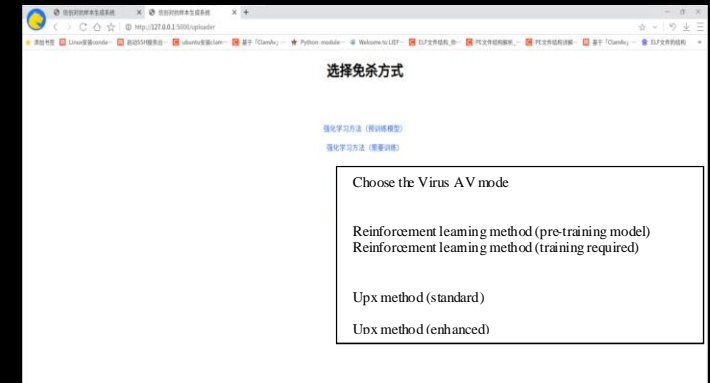
UPX is used to perform anti-detection operations on ELF files. Based on the standard UPX, an improved version called UPX with entry point obfuscation has been added. These two operations can achieve nearly 100% anti-detection rate against several antivirus software.

UPX method	Operation	Number of anti-detection using UPX + obfuscation	Number of UPX bypasses
upx_pack_mod1 ()	The improved version of UPX adds entry point obfuscation	Total number of samples is 100, successfully compressed 95, and the remaining 5 cannot be compressed due to their small size	95
upx_pack_mod2 ()	Standard UPX		

3.5 Solution Result

3.5.1 Prototype System for Automated Generation of Adversarial Malicious ELF Samples

Through encapsulation of APIs, a web-based prototype system for automated generation of adversarial malicious ELF samples is implemented. Below are screenshots of the system operation:



3.5 Solution Result

3.5.2 Experimental Results of Virus AV Methods Based on Reinforcement Learning

Figure 10. Detection rate of the sample before and after anti-detection processing by VT

CI	detection_rate
detection_rate	1/62
43/63	1/62
39/59	1/62
31/61	1/62
41/61	1/62
41/59	1/62
40/62	1/62
36/61	1/62
41/62	1/48
43/62	1/62
35/61	1/62
41/63	1/62
41/62	1/61
39/64	1/62
39/60	1/62
44/62	1/62
30/55	1/62
40/61	1/62
37/60	1/62
35/63	1/62
40/62	1/62
44/62	1/62
38/59	1/62
44/61	1/62
40/63	1/62
44/61	1/62
40/60	1/62
42/61	1/62
41/63	1/62
42/61	1/62

Figure 11. Example of process feature detection for the sample

```
Process
Behaviour: Execute a file
Detail info: execve: /home/nuaa/habo/virus/30_after/4fa4269b7ce44bfae5e574e6a37c38f_after
execve:
execve: -c ps -eo pid,cmd |awk '{print $2}' |grep -w deepin-diskmanager$
Behaviour: Process exit
Detail info: procexit status=0 ret=0 sig=0 core=0
procexit status=256 ret=1 sig=0 core=0
Behaviour: "Clone syscall, fork or vfork"
Detail info: clone: nil (PID=22189)
clone: nil (PID=22199)
clone: nil (PID=22210)
clone: nil (PID=22198)
clone: nil (PID=22209)
clone: nil (PID=22197)
clone: nil (PID=22208)
clone: nil (PID=22196)
clone: nil (PID=22195)
clone: nil (PID=22194)
clone: nil (PID=22203)
clone: nil (PID=22205)
clone: nil (PID=22191)
clone: nil (PID=22204)
clone: nil (PID=22190)
```

Figure 12. Example of file feature detection for the sample

```
File
Behaviour: File read
Detail info: read: path=/proc/764/status, size=1074
read: path=/proc/7743/cmdline, size=25
read: path=/proc/22195/status, size=1076
read: path=/proc/8/cmdline, size=0
read: path=/proc/7793/cmdline, size=50
read: path=/proc/12964/stat, size=184
read: path=/proc/2894/stat, size=313
read: path=/proc/19648/cmdline, size=20
read: path=/proc/2759/cmdline, size=0
read: path=/proc/19/cmdline, size=0
read: path=/proc/475/stat, size=158
read: path=/proc/2252/cmdline, size=122
read: path=/etc/gai.conf, size=0
read: path=/proc/17721/cmdline, size=0
read: path=/proc/853/cmdline, size=0
```

Figure 13. Example of network feature detection for the sample

```
Network
Behaviour: connect
Detail info: connect: 0->###916d264cc00 /var/run/ncsd/socket
connect: 192.168.174.134:45232->180.101.50.242:0
connect: 192.168.174.134:48024->101.37.43.178:1025
connect: 192.168.174.134:51733->180.101.50.188:0
connect: 192.168.174.134:56891->192.168.174.2:53
connect: 0.0.0.0->0.0.0.0
connect: 192.168.174.134:56248->192.168.174.2:53
connect: 192.168.174.134:60232->192.168.174.2:53
connect: 192.168.174.134:43555->192.168.174.2:53
connect: 192.168.174.134:38369->180.101.50.242:0
connect: 192.168.174.134:45212->192.168.174.2:53
connect: 192.168.174.134:45916->180.101.50.242:1025
connect: 192.168.174.134:52296->192.168.174.2:53
connect: 192.168.174.134:42248->192.168.174.2:53
connect: 192.168.174.134:52317->192.168.174.2:53
Behaviour: socket
Detail info: socket: domain=16 type=524291 proto=0
socket: domain=10 type=3 proto=58
socket: domain=10 type=2 proto=58
socket: domain=2 type=2 proto=1
socket: domain=2 type=528338 proto=0
socket: domain=16 type=528339 proto=0
socket: domain=2 type=3 proto=1
socket: domain=2 type=2 proto=0
socket: domain=1 type=528337 proto=0
socket: domain=2 type=524290 proto=0
```

3.5 Solution Result

Construction of the ELF

malicious sample database

725 X86 ELF viruses, 303 ARM ELF viruses, and 21 cross-platform compiled virus source codes.



AI-Based Generation

Technology for Adversarial

Malicious ELF Samples

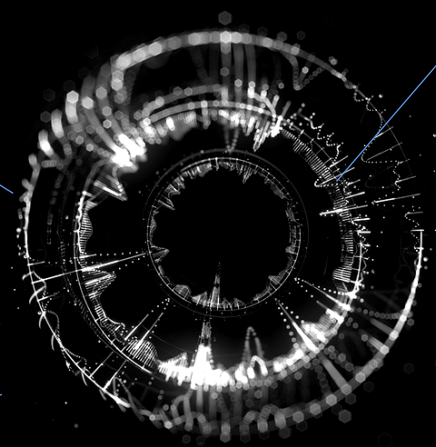
It has a bypass rate of 93%~100% against the current version of 360 Security Guard, ClamAV, and other detection engines.



Self-evaluation of the effectiveness

In comparison to the existing related technologies and bypass operations, this system has the following significant advantages in terms of bypass technology and effectiveness:

1. Innovative solution: AI-based, targeting domestic malicious ELF samples;
2. Leading effectiveness: With a high bypass rate, it is effective against lots of antivirus software;
3. Originality: It fills the gap in bypassing ELF.



3.5.3 Overall completion status

LLM: analysis, not generate

THANK YOU

**AI-Based Generation of Adversarial
Malicious ELF Samples**