Article

Horus: An Effective and Reliable Framework for Code-Reuse Exploits Detection in Data Stream

Gang Yang *, Xingtong Liu and Chaojing Tang

College of Electronic Science and Technology, National University of Defense Technology, Changsha 410073, China

* Correspondence: yanggang11@nudt.edu.cn.com

Abstract: Recent years have witnessed a rapid growth of code-reuse attacks in advance persistent threats and cyberspace crimes. Carefully crafted code-reuse exploits circumvent modern protection mechanisms and hijack the execution flow of a program to perform expected functionalities by chaining together existing codes. The sophistication and intrication of code-reuse exploits hinder the scrutinization and dissection on them. Although the previous literature has introduced some feasible approaches, effectiveness and reliability in practical applications remain severe challenges. To address this issue, we propose *Horus*, a data-driven framework for effective and reliable detection on code-reuse exploits. In order to raise the effectiveness against underlying noises, we comprehensively leverage the strengths of time-series and frequency-domain analysis, and propose a learning-based detector that synthesizes the contemporary twofold features. Then we employ a lightweight interpreter to speculatively and tentatively translate the suspicious bytes to open the black box and enhance the reliability and interpretability. Additionally, a functionality-preserving data augmentation is adopted to increase the diversity of limited training data and raise the generality for real-world deployment. Comparative experiments and ablation studies are conducted on a dataset composed of real-world instances to verify and prove the prevalence of *Horus*. The experimental results illustrate that *Horus* outperform existing methods on the identification of code-reuse exploits from data stream with an acceptable overhead. *Horus* does not rely on any dynamic executions and can be easily integrated into existing defense systems. Moreover, *Horus* is able to provide tentative interpretations about attack semantics irrespective of target program, which further improve system's effectiveness and reliability.

Keywords: intrusion detection; vulnerability exploit; machine learning; code-reuse attack; malware detection

1. Introduction

Identifying and defending against vulnerability exploits in the wild is an ongoing challenge. Although various protection mechanisms have been proposed, such as Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP) make it more difficult to exploit vulnerabilities, they can still be bypassed by carefully-crafted code-reuse attacks. Unlike ordinary shellcode-injection attacks, where external instructions are placed sequentially in memory, code-reuse attacks repurpose existing codes to perform specified computations, making the analysis and detection of exploits more difficult. Due to its versatility and concealment, code-reuse attack has increasingly become a mainstream means to generate control-flow hijacking exploits for memory corruption vulnerabilities.

Code-reuse attack is implemented by reusing existing assembly instructions in the memory space of the target software. Attackers can achieve the expected functionality by linking some certain instruction fragments (namely gadgets) chosen from the code section with executable permission of the target process. By leveraging code-reuse attacks, attackers are able to achieve arbitrary code execution without the injection of any code, and bypass DEP protections. Additionally, code-reuse attacks combined with memory disclosure techniques circumvent the widely-used protection mechanism of Address Space Layout Randomization(ASLR). Furthermore, various emerging methods are carried out after the release of Return Oriented Programming (ROP) [1,2], such as Jump Oriented Programming (JOP) [3], Block Oriented Programming (BOP) [4], etc. These methods extend the diversity of code-reuse attacks and also exacerbate the difficulty of prevention and detection against them.

Detection of code-reuse attacks relies on identifying their distinct characteristics from normal instances. One feasible countermeasure is to identify some specific runtime features distinct from normal executions. By monitoring the register or memory status, we can discover the anomalous control-flow jumps and violations related to the code-reuse attacks. While we are able to obtain an accurate detection with detailed information based on such dynamic methods, the heavy resource consumption and the requirement of prior knowledge on target program hinder their network-level usage and limit those methods within host-based application. In order to obtain real-time detection for code-reuse exploits explicit in data stream, static methods are more reasonable choices. Through statistical analysis or pattern matching based on statically inspecting the value characteristics of data stream, the local specificity can be figured out to facilitate the detection and location of code-reuse snippets hidden in consecutive bytes. Compared to dynamic methods, static methods can provide lightweight and universal countermeasures for code-reuse exploits detection. And static methods can be packed as a promising and portable component integrated into practical network-based defense systems. However, current static methods still face several challenges:

- First, current static methods suffer from high false positives and low effectiveness, especially during practical deployment due to the low signal-to-noise ratio.
- Second, massive static detection systems solely provide detection results without necessary explanations, especially the black-box property of learning-based method, cause an interpretable and readable problem, which hinders the further analysis on located suspicious bytes.
- Besides, publicly available code-reuse exploit samples are still scarce both in amount and diversity, which limits the practicality and generalization of learing-based methods, and causes severe performance degradation when handling unseen and evolving instances.

In this paper, we propose *Horus*, an effective and reliable static detection framework for code-reuse exploits hidden in real-world data streams, such as network traffic, to tackle those challenges stated above. We build a static detection model that combines time-series and frequency-domain analysis along with adversarial data augmentation. And an interpreter component is subsequently adopted to provide deterministic evidence of suspicious codes in order to make results more interpretable and readable. Consequently, we are able to obtain highly reliable and interpretable detection results against code-reuse exploits for further investigation during real-world application. Thus, the main contribution of the paper is as follows:

- We propose a novel static detection model with collaborative analysis of time-series and frequency-domain characteristics of code-reuse codes in the byte level, which can largely alleviates the adverse effects caused by background noise bytes and raise the signal-to-noise ratio to enhance the model's effectiveness.
- We innovatively introduce a corpus-based interpretation mechanism to enhance the interpretability and readability of the outputs from our framework. We feed the suspicious bytes into a corpus-based interpreter module to tentatively recover the hidden attack semantics, which can provide a preliminary explanation for further investigation and forensics.
- Additionally, we adopt a functionality-preserving data augmentation strategy to incrementally generate feasible exploits for the extension of training data. By expanding the original dataset with variant samples, we can increase the diversity of training data, which will greatly improve the generalization and practicality of our framework.

2. Background and Related Work

In this section, we first briefly introduce the background knowledge of code-reuse exploits and the technical principles of code-reuse attacks. Then we comprehensively review and compare the current detection methods for code-reuse exploits in various dimensions to highlight the strength of our proposed method.

2.1. Code-reuse Exploits

The manipulation of vulnerability exploits has been evolving along with the growth of defensive techniques in sophistication. Generally, early control-flow hijacking attacks consists of two steps. The attacker would first inject shellcode that performs attacker's desired behaviors such as spawning a shell, into the memory space of the target program. Then the attacker exploits a vulnerability to redirect the control flow to the injected shellcode. At this point, the attacker could execute arbitrary instructions in the context of the vulnerable program. However, direct code injection attacks have been largely eliminated due to the widespread adoption of W$\oplus$X security mechanism such as DEP and NX. As a typical implementation of W$\oplus$X security mechanism, DEP marks all user-controllable memory regions with an attribute value as non-execution. It means that the attacker cannot directly turn the control flow to the externally injected codes after control flow hijacking. The attacker is only allowed to restrict the control flow within the memory regions with executable property. Code-reuse exploit is a technique for exploiting a security vulnerability by chaining together unintended instruction sequences already present in the memory space of the compromised program. The instruction sequences, also called gadgets, are carefully chosen from ".data" section of the binary loaded in the memory space according to some certain rules. Because the memory regions where the gadgets located are inherently marked as executable, the malicious codes implemented by the gadgets can effectively bypass the executable-space protection. Considering the fact that executable protection has been adopted as a basic security feature integrated modern software systems, code-reuse attacks have become universal approaches to circumvent mitigations implemented in modern operating systems. Although emerging mitigation mechanisms such as ASLR (Address space layout randomization) are proposed to increase the difficulty of vulnerability attacks, code-reuse exploits are still effective measures with various evolvingly-spawned schemes. For instance, information leakage operations can be added to the gadget chain to infer code location. And then the carefully crafted exploits can be generated to bypass the ASLR mechanism. Besides, the proven Turing Complete of code-reuse attacks also make them considered as the primary attack technique of choice for nearly all modern exploits of memory safety vulnerabilities.

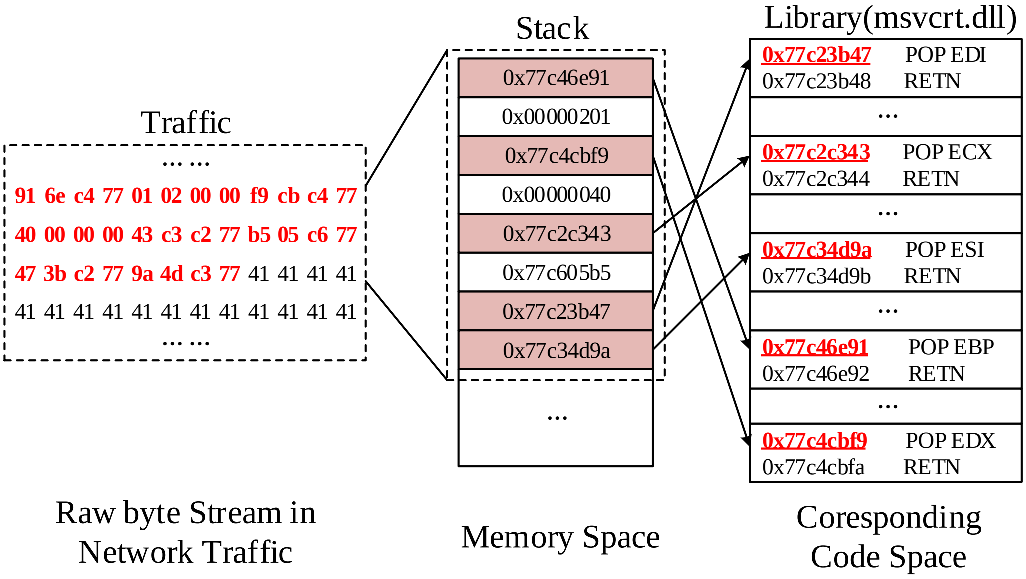


Figure 1. A typical example of code-reuse exploit with ROP techinique.

A representative code-reuse exploit which leverages the ROP technique can be shown in Figure 1. In the view of composition, the exploit is composed of gadget addresses and immediate operands. From the perspective of dynamic runtime characteristics, the execution of compromised program would deviate from the expected control flow. Each gadget can be considered as a tiny function, and attackers chain the gadgets by multiple entry addresses and "ret" instruction. The attacker is able to reuse and chain the codes already exist in the program's memory space to execute arbitrary code by manipulating the return addresses.

Despite the fact that the composition of code-reuse fragments varies according to the concrete vulnerability environments, some common characteristics remain in both dynamic domain and static domain. Hence, those commodities are used as distinguishing features to detect underlying code-reuse exploits.

2.2. Detection Methods of Code-reuse Exploits

Various methods have been proposed to identify the code-reuse exploits on the foundation of commodities that exist in current attack schemes. Those methods can be categorized into dynamic and static methods, depending on whether the transmitting data is required to be fed to the corresponding program for execution.

2.2.1. Dynamic methods

Under dynamic methods, researchers distinguish the code-reuse exploits from benign data by leveraging the runtime information acquired from execution environment. It requires researchers to precisely rebuild the target execution environment to ensure that the exploits can be performed as intended. The basic principle and assumption of dynamic methods is that there will be some anomalies triggered in CPU registers or memory once exploits executed. Therefore, hardware and software technologies are employed to perform checks on CPU registers or memory to collect related clues from a running system.

Some researchers proposed to monitor the sequence of instructions executed by a program and flag instruction streams with a high frequency of "ret" operations as exploits [7–11]. For instance, DROP [7] inspects instruction sequences ending in "ret" operation based on dynamic binary instrumentation. Then we can check whether ROP malicious code exists by comparing the number of instructions in each gadget-like sequence and the length of contiguous gadget-like sequences with defined thresholds separately. Unexpected jumps of instructions can also be utilized as an important identifier of code-reuse attacks [8,9]. Ropecker [10] relies on hardware-assisted LBR (Last Branch Recording) to identify the indirect branch instructions to determine whether there exists an excessively long series of gadget-like instruction sequences. Xede [11] monitors the presence of unexpected returns based on a whole-system virtualization platform and raises alerts when there are N consecutive mismatched return addresses.

CFI (Control Flow Integrity)-based methods keep track of the control flow transfer of the target program. Those methods verify the legitimacy of each control flow transfer by checking whether the flow transfer exists in the legitimate control flow graph sets. Ropdefender [12] and ROP-Hunt [13] instruments all return instructions issued during program execution and checks the existence of code-reuse attacks by recognizing the control flow corruptions. Argos [14] employs dynamic taint tracing to examine whether the EIP register is tainted by input data. The literature [15] presents a novel scheme named code shredding for detecting invalid control-flow transfers based on address checksums. However, the software-based CFI always cause huge resource consumption. To mitigate the heavy computation overhead, CFIMon [16] leverages the branch tracing store mechanism of modern processors to collect runtime traces and detect violation of control flow integrity. kBouncer [17] monitors indirect branch events to discover abnormal control transfers. RipRop [18] counts the indirect branches and produces alarms when the counter exceeds a threshold. ROPStop [19] monitors and verifies the program counter and callstack based on the Dyninst binary modication toolkit.

Pattern matching on some runtime features is an alternative strategy can be taken to hunt code-reuse exploits. SCRAP [20] defines attack signatures based on runtime features from hardware, and then performs a dynamic detection of code-reuse attacks. EigenROP [21] builds an unsupervised learning-based detector that takes advantage of architecture-agnostic statistical characteristics of the program, such as memory reuse distance, register traffic load, and memory locality gathered from snapshots. ROPSentry [22] detects ROP attacks at runtime by observing different hardware events between which ROP exploits trigger and normal programs.

Memory forensics is also an effective way to recognize code-reuse gadgets for revealing the existence of code-reuse attacks. The literature [23] focuses on analyzing code reuse attacks embedded in various document formats. Authors scan the data regions of the snapshot to find gadget-like pointers and statically profile the behavior of those gadget-like codes. UnROP [24] scans ROP chains in a process memory dump and verify its functionality. According to the scanned input data, Ropscan [25] speculatively drives the execution of code that already exists in the address space of a targeted process and detects the execution of ROP code at runtime. In literature [26], authors design and implement deRop to automatically convert ROP-encoded shellcodes into original shellcodes and verify its semantic validness. ROPMEMU [27] performs emulation over a memory dump and recovers their precise control flow graphs to examine the existence of ROP chains. deExploit [28] focuses on the identification of overwritten data structure in memory to diagnose the memory corruption exploits

2.2.2. Static methods

In contrast to dynamic methods, static methods detect code-reuse exploits by directly analyzing streaming data without execution. From empirical views, the value range and arrangement of code-reuse exploits obey some specific properties and implicit patterns. Those patterns can be acquired or discovered from statistics or human knowledge. Since researchers are free from the efforts to precisely reproduce targeted environment for exploits, static methods receive increasing attention both in academic and industrial communities.

Based on the intuition that packets with code-reuse exploits contain the addresses of instructions or library functions, there will be statistical byte-level anomalies in those packets. In literature [30], Ho introduces a lightweight method based on the Kullback-Leibler divergence between the probability distributions of the normal packets generated from IoT devices and the abnormal packets containing gadget addresses. Furthermore, Ho proposes an improved scheme [31,32] by incorporating the Sequential Probability Ratio

Test (SPRT) or Sequential Hypothesis Test (SHT) with the probabilistic inspection on the packets incoming into industrial IoT devices. EavesROP [33] uses a sliding window to locate potentially suspicious data segments and applies Fast Fourier Transform-based Pattern Matching to compare the suspicious address values present in the network data stream with the actual Gadget addresses in known library files. And we can determine the occurrence of potential ROP attacks based on whether the output exceeds the threshold.

Machine learning methods are also applied to mastering the underlying numerical patterns. Researchers train a convolutional neural network (CNN) or reccurent neural network (RNN) respectively based on the customized dataset consists of benign samples and crafted gadget chains [34–36]. ROPminer [37] statically detects ROP chains by learning the orders of ROP components and the byte patterns of each component. The authors build a HMM (Hidden Markov Model) model based on exhaustively collecting feasible ROP gadgets in the target libraries. STROP [38] proposes a series of rules to classify and mark suspicious bytes with different granularity according to the regular characteristics of the range of values taken by gadget addresses and further extracting 7-dimensional statistical characteristics to achieve judgment. In literature [39], authors introduce the definition of RCI (ROP chain integrity). Authors detect ROP-based malicious documents by calculating the likelihood ratio to malicious or benign samples considering the RCI.

Prior knowledge provides a corpus of candidate payload bytes which are widely used in code-reuse attacks. Those specified bytes can also work as identifiers to determine the existence of code-reuse attacks. Signature-based detectors [40] such as Nebula leverage the byte pattern composed of some specific bytes have been widely used for detecting known vulnerability exploited. The literature [41] presents a prototype focusing on the detection of *sledge* pattern used in HTTP requests. Tanaka et al. propose n-ROPdetector [42] to achieve the matching of network streams by collecting gadget patterns from the Metasploit framework.

Emulation on suspicious data is also a feasible approach for statical detection of code-reuse exploits. The literature [43] introduces Code-Stop to determine whether there is a code reuse attack by simulating the execution of a potentially suspect gadget address and comparing whether its semantics are similar to the assignment pattern of registers when calling specific Windows functions (e.g., *VirtualAlloc()*, *SetInfoProcess()*, *VirtualProtect()*). The literature [44] proposes Ropdissector to reduce the workload of subsequent analysis by simulating the execution of suspicious gadgets and being able to further distinguish and label the semantic functions of ROP fragments.

2.2.3. Comparative analysis and motivation

In summary, dynamic methods and static methods show their strengths and weaknesses, respectively.

Generally, dynamic methods focus on the deterministic properties of program behaviors when code-reuse exploits are being performed. Therefore, dynamic methods can provide precise detection with low false alerts and auxiliary information for further diagnosis. Additionally, dynamic methods adapt to the detection on encoded or obfuscated exploits, since the exploits will be recovered when executed. However, the assumption or prerequisite of dynamics methods is that the exploits will be successfully injected into memory and executed. Researchers should prepare the corresponding analysis environments in advance. As is known, it is difficult to accurately reveal an exploit's targeted environment and perfectly recover analysis environment with same configurations. That means numerous exploits hunted from network will not be activated due to the lack of prior knowledge about targeted environments, incurring possible false negatives on real-world deployment. Worse still, the trade-off between monitoring granularity and resource consumption is another significant factor which hinders the piratical application of dynamic methods. And the heavy resource consumption will reduce the throughput of whole system and hinder researchers in obtaining immediate results.

On the other hand, static methods provide a lightweight and generic solution on code-reuse exploits. Compared to dynamic methods, static methods hold higher throughput and easy-to-deploy characteristics for network-level application. We can capture and scrutinize all suspicious data from network traffic without efforts to construct an analysis environment and prior knowledge about influenced software. Current static methods mainly employ stochastic analysis or learning-based algorithms from data-driven insight, which focus on data characteristics of code-reuse exploits. That means static methods are largely influenced by noises from background bytes and distribution drifts in unseen data, leading to massive false-positives. And most of static methods cannot provide deterministic evidence or visual interpretation to support detection decisions and further investigations. Additionally, limited publicly available samples also exacerbate performance degradation during real-world applications. Although the shortcomings stated above restrict the reliability and interpretability of static methods, static methods remain competitive approaches owing to the development of machine learning methodology. The versatility and practicability make it a promising direction worthy of further study and exploration.

Considering the strengths of static methods in handling network data traffic, we propose a novel framework on the basis of previous work. Our framework carries forward the advantages of static methods and attempts to remedy shortcomings by providing a more effective, reliable, and interpretable solution. We synthesize the complementary views from time series and frequency domain to build our learning-based system in order to reduce the influence from background noises and increase the reliability of decisions. And we leverage a function-preserving data augmentation to extend the diversity of existing training data, help our model get beyond local overfitting and raise the transferability and generality of our method. Furthermore, we integrate the corpus-based translation on the basis of lactation of suspicious bytes, to achieve extremely lightweight speculative interpretation. The auxiliary information from interpreter provides deterministic evidence to support further investigation. It means that, through the corpus-based interpreter, we can enhance the readability and interpretability of output and overcome the black-box problem to a certain extent.

3. Problem Statement and Theoretical Analysis

In this section, we present a problem formulation and theoretical analysis on static code-reuse exploits detection, which provides support for the design of our framework.

From the perspective of static analysis, detecting code-reuse exploits in data stream can be formulated as identifying subsequences with certain value characteristics. The value characteristics are derived from sequential layout of gadget addresses mixed with const parameters. Considering that the const parameters do not show enough recognition in numerical value, we regard the const parameters as noises and concentrate on characterizing the gadget parts of a subsequence. We use chunk to represent a couple of consecutive bytes with the same size as a gadget. In order to take advantage of the value characteristics from gadget address, we divided the entire sequence into chunks with fixed length depend on whether the architecture of targeted environment is 64-bit or 32-bit.

To avoid confusion, all elaborations in this paper are conducted under a 32-bit enviroment by default, since in a 64-bit system environment we only need to double the sampling channels.

Based on the above analysis, we scrutinize the value characteristics of a sequence in two dimensions:

- The value characteristics of single chunk are determined by the gadget $g \in \{\mathcal{G}\}$.
- Meanwhile, the value characteristics of consecutive chunks are determined by arranged gadgets as $[g_0 g_1 g_2 ... g_{p-1} g_p ... g_n]$.

In the view of a single chunk, the values of a single chunk are mainly restricted by the address values of feasible gadgets $g_i \in \{\mathcal{G}\}$. Gadgets are strictly chosen from some library regions of memory space and occupy specified functionalities such as arthritic operation, register move, memory read or write, and instruction pointer jump.As a result, the amount of gadgets which satisfy the requirements are usually enumerable in practice. Especially, some gadgets from common libraries might be broadly used in various vulnerability exploits which share similar runtime environments, such as the "universal gadgets" chosen from the *_libc_csu_init* function and some gadgets from commonly-used DLLs. For example, "*0x77C34fCD, # POP EAX # RETN*" from *msvcrt.dll* are broadly used in exploits of *EDB-ID-27012*, *EDB-ID-23785*, *EDB-ID-28187*, and *EDB-ID-37056*, etc. It means that there might be seen address values laying in different code-reuse exploits. This commonness can be used as one of the theoretical fundamentals for stochastic detection.

On the other hand, even though attackers employ alternative gadgets from those common libraries, the addresses of those gadgets take value in the range of source libraries. For randomized processes, the gadget space is even smaller, attackers often pick gadgets from a few coommon non-ASLR DLLs such as *msvcrt.dll*, *hxds.dll*, and *msvcr71.dll* to reduce the difficulty of exploiting. As the statistical data in the literature [38] stated, the sizes of over windows 90% DLLs is smaller than 1MB, while the sizes of over Linux 90% .so files is smaller than 128KB. For example, *msvcr71.dll* is always loaded in virtual memory region from *0x7C34000* to *0x7C396000*. Therefore, the ROP gadget addresses from *msvcr71.dll* are in the range of *0x7C34000-0x7C396000*. It means that the address value of gadget varies within a limited range, and the high byte value of address might not change. Or worse, even if attackers use gadgets from program-specific DLLs, there will still be on excessive difference in address values since all the DLLs are loaded into almost the similar memory region. For example, in term of Internet Explorer(IE) 9 of Windows 7, 124 DLLs are loaded, of which 30 DLL files are loaded in *0x6XXXXXXX* while 94 DLL files are loaded in *0x7XXXXXXX*.

In the view of chunk sequences, values are determined by the ingredients and orders of gadgets $[g_0 g_1 g_2 ... g_{p-1} g_p ... g_n]$. To accomplish some functionalities during vulnerability exploits, attackers borrow some existing instructions and reorganize them in a certain order to achieve the corresponding attack semantics. Owing to the address-to-gadget corpus in the target program, the gadget chains are translated into the form of address sequences written in exploits. Hence, the linkage and composition of the gadgets in turn further determine the layout and arrangement of the corresponding address values. And due to the semantic connotations from the intended gadget chains, the values of chunk sequences naturally follow some resembled and transferable paradigms. We use the local fluctuation of address values in chunk sequences to sketch those paradigms, which are perceptible in many real-world exploits.

The numerical characteristics in two dimensions form the basis of our study. During the detection phase, the code-reuse exploits are hidden in network traffic mixed with protocol text bytes, application-specific bytes, plain text bytes, and encoded text bytes. The core idea is to enhance the distinction and relief the influence from those noises by applying feature extraction and enhancement algorithm. Concretely, we leverage the complementary time-series and frequency-domain features to enhance the signal-to-noise ratio of code-reuse exploits against background noises. Next, we adopt data augmentation to improve the generalization of detectors and combat the local overfits from a limited volume of available data. Additionally, we employ a corpus-based interpreter to provide possible deterministic evidence and interpretability of detection.

4. Methodology

In this section, the implementation of the proposed framework are presented in detail. First, we introduce the overall design of *Horus*, an interpretable joint time-frequency detection framework with data augmentation. Then we will demonstrate the implementation details of each module in our framework.

4.1. Overall Framework

An overall architecture of the proposed *Horus* framework is shown in Figure 2. *Horus* framework contains four core modules: preprocessing and sampling module, joint time-frequency detection module, function-preserving data augmentation module, and corpus-

doi:10.20944/preprints202209.0286.v1

based interpreter module. For incoming data stream, we conduct basic preprocessing and sampling to convert incoming data into the form of fixed-length sequences. Next, we feed the sequences into the joint time-frequency detection module to obtain decision results. Finally, we employ a corpus-based interpreter to tentatively translate the suspicious bytes to instructions and acquire deterministic evidence that enhance the interpretability and convincingness of detection results. In addition, in order to increase the diversity of training data, we utilize data augmentation technology to generate polymorphic variants from original samples.

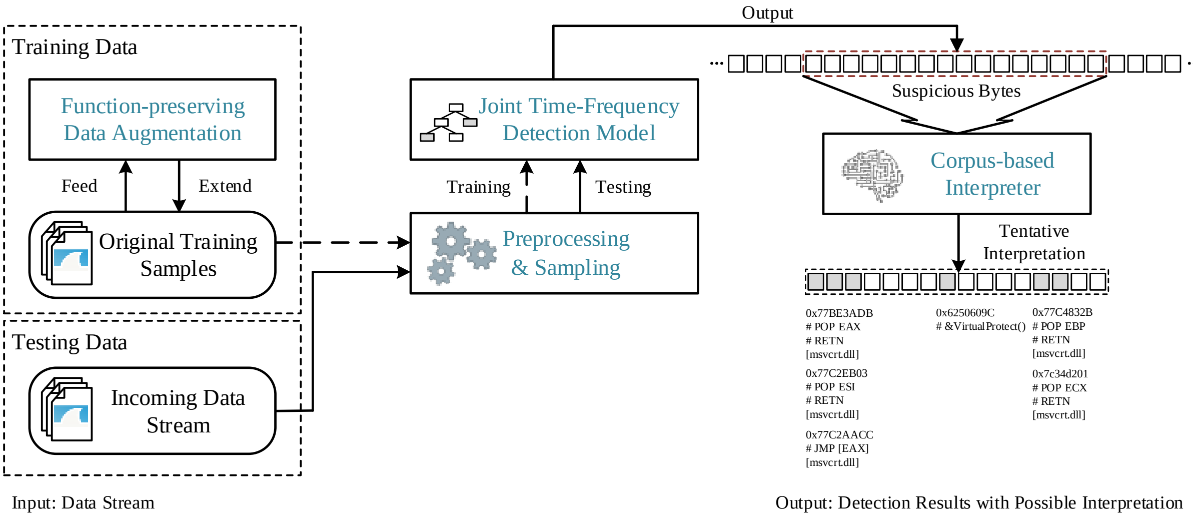


Figure 2. An overview of *Horus* framework.

4.2. Preprocessing and Sampling

The preprocessing and sampling module is responsible for transforming data stream into fixed-length data segments. Then we adopt a sliding windows of fixed length to sample from retrieved payload as illustrated in Figure 3.

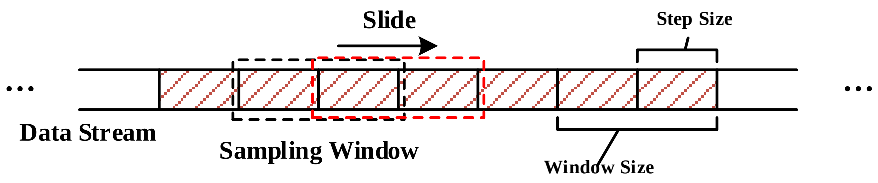


Figure 3. Sampling based on a sliding window.

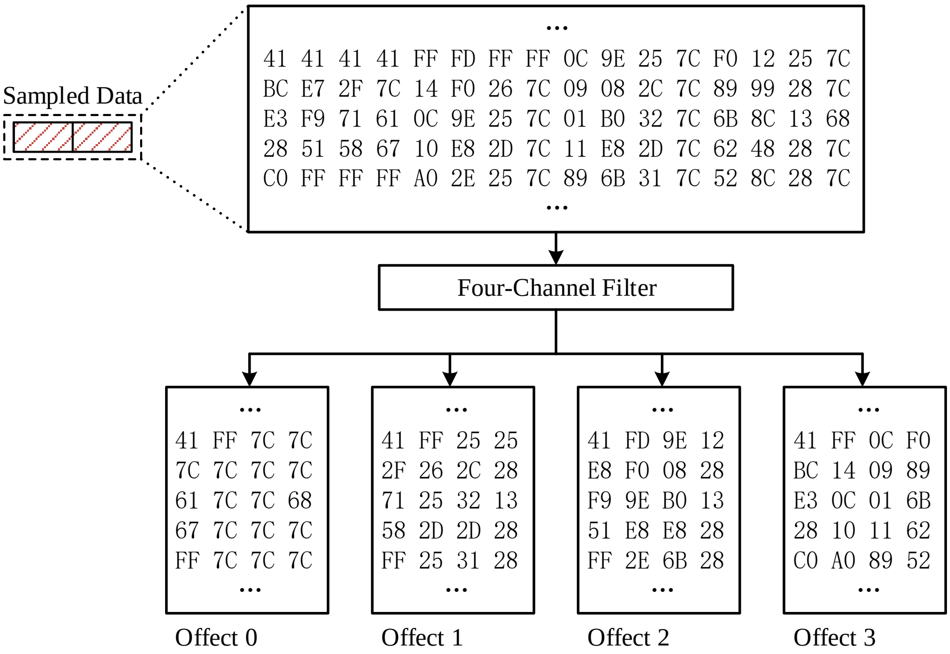
Notice that we use bytes as the unit for checking data stream. We assume that the maximum size of usual code-reuse exploits is L in bytes. To avoid possible overlap and truncation during data sampling, we set the size of sliding-window as $2L$ and the step size as L. Therefore, we can guarantee that there exist at least one window cover an entire code-reuse payload during sampling. In our design, we set step size L as 256, so that the window size is 512.

4.3. Joint Time-frequency Detection Model

In the detection module, we incorporate discriminative features in both time series and frequency domain to build an ensemble classification model. Our model inherits the advantages of time-series and frequency-domain analysis, is promising to exhibit a noise-tolerant performance on detection tasks.

4.3.1. Time-series feature extraction

Considering the variations of byte values in different gadgets differ from high to low positions. Next, we use a multi-channel filter recompose the sampled data into four sequences to sketch the value fluctuation features, respectively. Notice that our approach is assumed to conduct under 32-bit environment by default. In 32-bit architecture, a payload consists of several four-byte aligned addresses, then a four-channel filter is suitable. The offset of a code-reuse payload in the data buffer is unknown, but we know that each address is four bytes and addresses are aligned inside a payload. Therefore, we create four sequences, one for each offset, see Figure 4.



```
...
41 41 41 41 FF FD FF FF 0C 9E 25 7C F0 12 25 7C
BC E7 2F 7C 14 F0 26 7C 09 08 2C 7C 89 99 28 7C
E3 F9 71 61 0C 9E 25 7C 01 B0 32 7C 6B 8C 13 68
28 51 58 67 10 E8 2D 7C 11 E8 2D 7C 62 48 28 7C
C0 FF FF FF A0 2E 25 7C 89 6B 31 7C 52 8C 28 7C
...
```

```
Offect 0      Offect 1      Offect 2      Offect 3
...           ...           ...           ...
41 FF 7C 7C   41 FF 25 25   41 FD 9E 12   41 FF 0C F0
7C 7C 7C 7C   2F 26 2C 28   E8 F0 08 28   BC 14 09 89
61 7C 7C 68   71 25 32 13   F9 9E B0 13   E3 0C 01 6B
67 7C 7C 7C   58 2D 2D 28   51 E8 E8 28   28 10 11 62
FF 7C 7C 7C   FF 25 31 28   FF 2E 6B 28   C0 A0 89 52
...           ...           ...           ...
```

Figure 4. Channel decomposition before time-series feature extraction.

From the time-series perspective, we focus on local fluctuation of address values in code-reuse payloads. Derived from the similar instruction semantics and source libraries, there exist some common features in value fluctuation which reflect the paradigms of gadgets arrangement. The regular fluctuation of byte values embodies the overall ordering characteristics of code-reuse gadget chains. And the dispersed segments in code-reuse payloads that comply with the fluctuation features can support the decision. As elaborated in Section 3.1, local fluctuation of values can be used as a distinct identifier of the gadget chain against noises. And this idea can be thought as an ameliorative version of previous work [31].

doi:10.20944/preprints202209.0286.v1

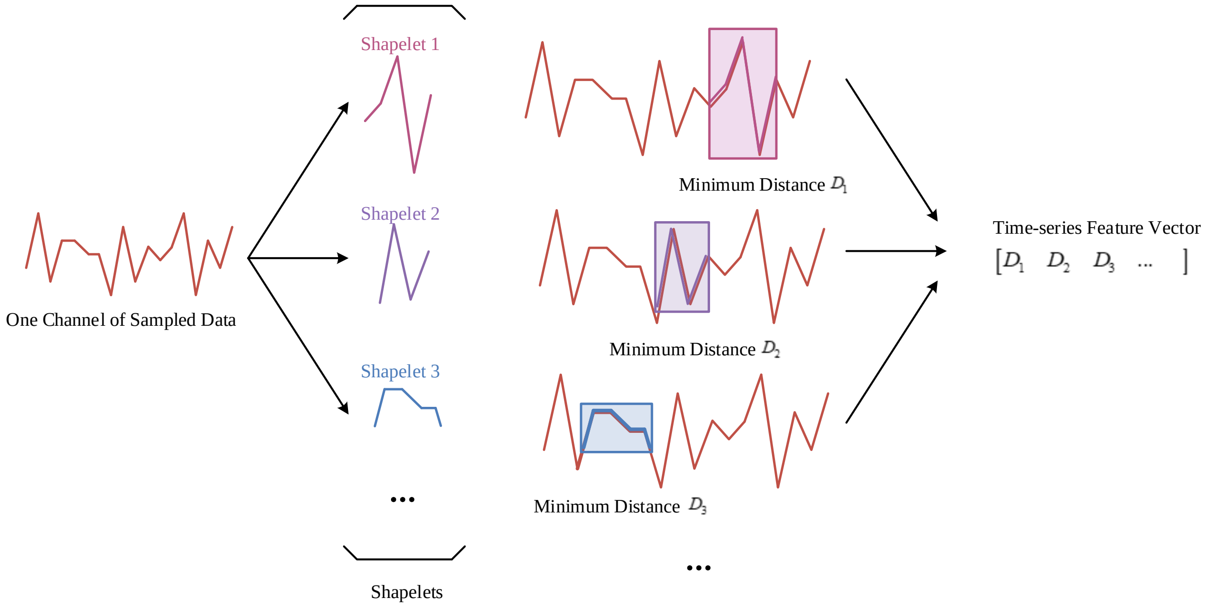


Figure 5. Time-series features extraction from *Shapelet Transform*.

In implementation, we leverage the *Shapelet Transform* algorithm to extract the time-series fluctuation features of address values to sketch those paradigms. The workflow of *Shapelet Transform* is shown in Figure 5. The subsequence S with varying length L is defined as a subset of continuous values from the entire time-series sequence. A shapelet is defined as one of the most discriminating time-series subsequences used to describe local value fluctuation of a sequence. We select the subsequence with distinct shape feature which is common in sequences from training set and then list it as a candidate shapelet. Then we use the shapelets as a set of basis vectors that represent the most discriminative fluctuation patterns for vectorization of incoming time-series data. For each channel, we collect 32 distinct shapelets to work as basis vectors. During *Shapelet Transform*, a sequence is described by its similarity(minimum distance) to each of a collection of shapelets as $[D_1, D_2, D_3, ..., D_L]$. In general, after *Shapelet Transform*, each sequence is represented by a vector of 32 dimensions describing how similar it is to each of the 32 selected shapelets. And the sampled data can be represented by a 128-dimension vector.

4.3.2. Frequency-domain feature extraction

From the frequency-domain perspective, we aim to mine the implicit cyclical patterns of byte values in the global sight. Through frequency-domain analysis, the cyclical patterns can be decomposed into different frequency components. The aggregation paradigms of frequency components depict common features of entire code-reuse payloads.

In this work, we utilize *Discrete Wavelet Transform* (DWT) to implement frequency-domain decomposition and extract the structure features on windowed non-stationary data. *Discrete Wavelet Transform* works on discrete non-differentiable spaces, and is capable of handling signals with sudden transitions. In practice, we choose *Discrete Haar Wavelet Transform* (DHWT) in our design. As the simplest form of wavelet transforms, *Discrete Haar Wavelet Transform* outperforms traditional Fourier transform on sketching frequency-domain characteristic of a local data segment, and easier to be implemented and integrated into existing systems. By using *Discrete Haar Wavelet Transform*, it is possible to detect code-reuse payloads even in the presence of noise and assuming that the target system employs ASLR. In mathematics, the *Haar wavelet* is a sequence of rescaled “square-shaped” functions that together form a family or basis of wavelets. During the process of decomposition, two types of functions play an important role. The wavelet function $\psi(\text{t})$ and the scaling function $\phi(\text{t})$ is defined as follows:

$$\psi(t) = \begin{cases} 1, & 0 \leq t < \frac{1}{2} \\ -1, & \frac{1}{2} \leq t < 1 \\ 0, & t < 0, t > 1 \end{cases} \tag{1}$$

$$\phi(t) = \begin{cases} 1, & 0 \leq t < 1 \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

Before decomposition, we must craft a series of basis functions $\{\varphi_{j,k}(n)\}$ and $\{\psi_{j,k}(n)\}$ based on the wavelet function and the scaling function, as described in following equations:

$$\varphi_{j,k}(t) = 2^{j/2}\varphi\left(2^{j}t - k\right) \tag{3}$$

$$\psi_{j,k}(t) = 2^{j/2}\psi\left(2^{j}t - k\right) \tag{4}$$

where j indicates the number of transform levels, and k is the position parameter.

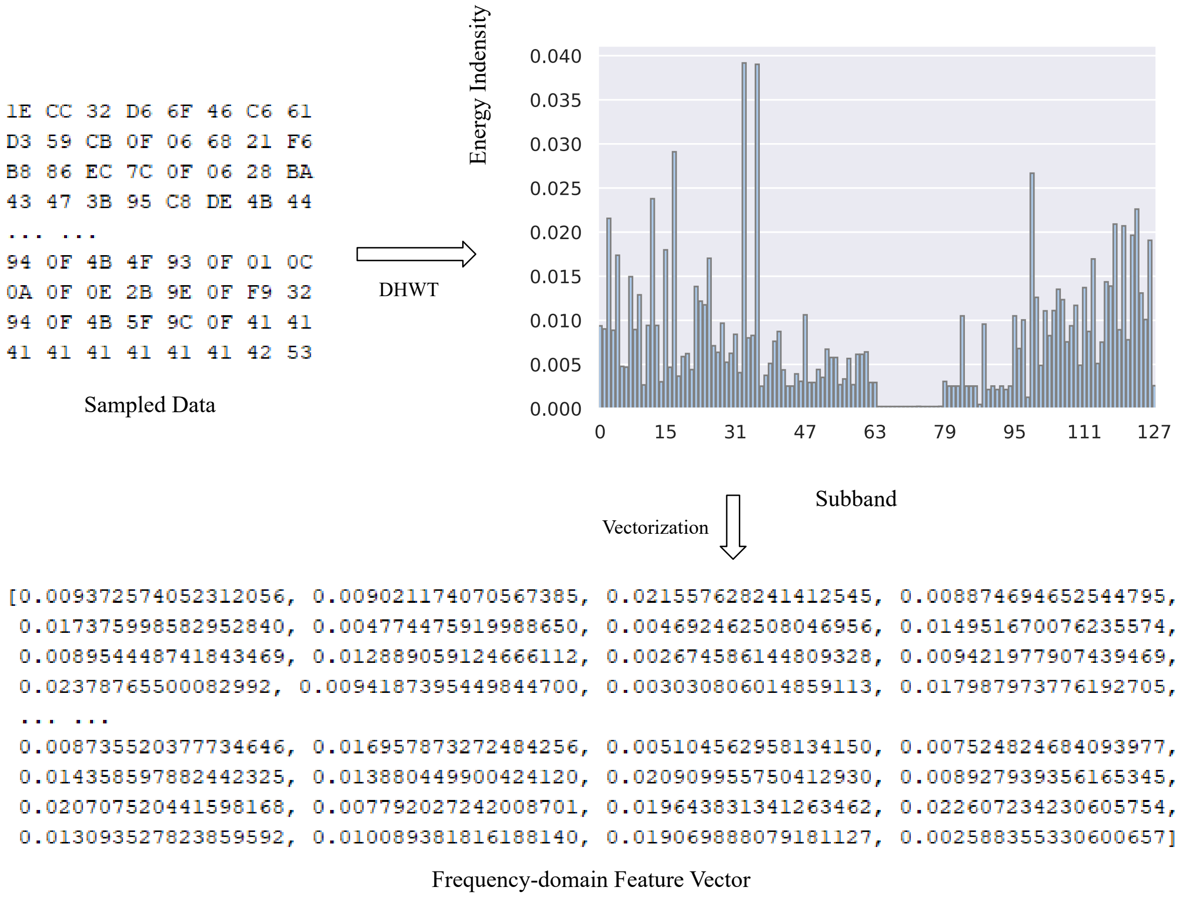
In this study, we conduct *Discrete Haar Wavelet Transform* at the first level to obtain approximation and detail coefficients. We focus on approximation coefficients produced by the transformer, which represents the coarse-scale, low-frequency components of a signal. For an input sequence $s(t)$ which consists of T positions, approximation coefficient is defined as:

$$a_{j,k}(t) = \left\langle s, \phi_{j,k} \right\rangle = \sum_{t=1}^{T} s(t)\phi_{j,k}(t) \tag{5}$$

To obtain appropriate feature representation, we split the frequency band into 128 sub-bands. We further calculate the energy indensity according to the following equation and build the corresponding histogram of energy spectrum.

$$\rho(t) = \sum^{\Delta t} |a_{j,k}(t)|^2 \tag{6}$$

The process of frequency-domain features extraction is illustrated in Figure 6.



```
1E CC 32 D6 6F 46 C6 61
D3 59 CB 0F 06 68 21 F6
B8 86 EC 7C 0F 06 28 BA
43 47 3B 95 C8 DE 4B 44
... ...
94 0F 4B 4F 93 0F 01 0C
0A 0F 0E 2B 9E 0F F9 32
94 0F 4B 5F 9C 0F 41 41
41 41 41 41 41 41 42 53
```

```
[0.009372574052312056, 0.009021174070567385, 0.021557628241412545, 0.008874694652544795,
 0.017375998582952840, 0.004774475919988650, 0.004692462508046956, 0.014951670076235574,
 0.008954448741843469, 0.012889059124666112, 0.002674586144809328, 0.009421977907439469,
 0.02378765500082992, 0.009418739544984470O, 0.003030806014859113, 0.017987973776192705,
 ... ...
 0.008735520377734646, 0.016957873272484256, 0.005104562958134150, 0.007524824684093977,
 0.014358597882442325, 0.013880449900424120, 0.020909955750412930, 0.008927939356165345,
 0.020707520441598168, 0.007792027242008701, 0.019643831341263462, 0.022607234230605754,
 0.013093527823859592, 0.010089381816188140, 0.019069888079181127, 0.002588355330600657]
```

Figure 6. Frequency-domain features extraction from *Discrete Haar Wavelet Transform*.

The y-axis of the histogram represents the cumulative intensity of a given frequency component on the x-axis. We utilize relative values instead of the exact values in the diagram. Then we convert the energy spectrum into tabular data and acquire a 128-dimension vector which represents the frequency-domain characteristic. Moreover, through frequency-domain analysis, the inference from noise bytes can be largely reduced, and the signal-to-noise ratio is promisingly raised.

4.3.3. Architecture of detection model

Generally, time-series and frequency-domain features work as complementary features for accurate code-reuse exploit detection on windowed data. Shapelet features illustrate the local similarity of sequencing features and eliminate the noises from unrelated bytes lying in data streams. However, the shapelet features are sensitive to data with very small variations. It means that small noises located on the internals might cause mismatchs between incoming sequences with shapelets, leading to false negatives in practical application. Meanwhile, *Haar Wavelet Transform* is insensitive local value variations, which plagues shapelet-based methods. The overall frequency-domain component characteristics can be well represented, unaffected by local noises. Mixed noise can be effectively eliminated by frequency-domain filtering. On the other hand, that means the subtle common fluctuations in which shapelet-based methods specialize might also be discarded by those frequency-domain methods. Totally, the complementarity of time-series and frequency-domain methods make it promisingly effective by integrating both time-series and frequency-domain features. Therefore, a joint time-frequency models are logical to be proposed.

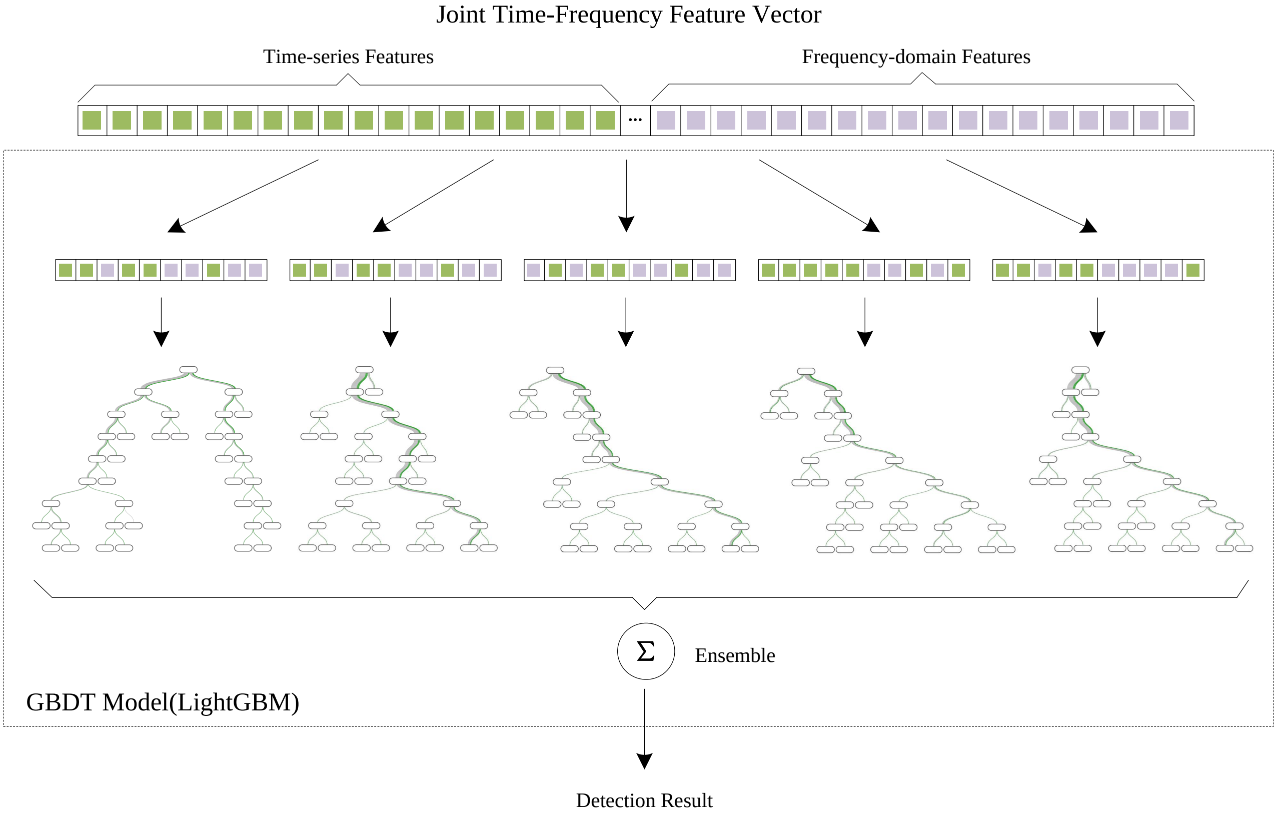


Figure 7. Architecture of GBDT-based detection model.

The architecture of the joint time-frequency model is illustrated in Figure 7. After feature extraction, the sequences are transformed into a 256-dimension tabular vector composed of 128-dimension time-series features and 128-dimension frequency-domain features. Empirically, tree-like models work better than other machine-learning models on highly structured datasets, such as tabular data. Therefore, we employ a tree-based ensemble model named Gradient Boosting Decision Tree (GBDT) to construct the classifier model. GBDT uses decision trees which work on different parts of features as base classifiers. Then, as an ensemble model, GBDT adopt boosting algorithm to integrate multiple "weak" base classifiers into a "strong" one. When training, the optimization is to decrease the sum errors of base classifiers.

In implementation, we employ a highly efficient framework named LightGBM to build the ensemble classifier. As an improved variant of GBDT algorithm, LightGBM combines the predictions of multiple decision trees to make the final prediction generalize well. In LightGBM, two exclusive strategies are integrated, namely Gradient-based One-Side Sampling (GOSS) and Exclusive Feature Bundling (EFB), which bring prevalent performance on accuracy and computational efficiency. On selection of hyperparameters, we leverage the Grid-search Algorithm 1 to sort out the best hyperparameters. Under our experimental configuration, we find the optimal hyperparameters are $\{'num\_leaves' : 31, 'subsample' : 0.8, 'feature\_fraction' : 0.9, 'learning\_rate' : 0.02\}$.

Algorithm 1: Grid-search Algorithm

```
Input: hyperparameter set $P = \{p_1, p_2, p_3, ..., p_m\}$, each hyperparameter values in
       a series of candidate values as $p_k \in [p_k^{(i)}]$
for $p_1$ in $[p_1^{(i)]}]$ do
    for $p_2$ in $[p_2^{(i)]}]$ do
        for $p_3$ in $[p_3^{(i)]}]$ do
            ...
            for $p_m$ in $[p_m^{(i)]}]$ do
                $model = ligtgbm.train(training\_set, params = \{p_1, p_2, p_3, ..., p_m\})$
                $score = model.predict(valid\_set)$
                $cv\_list.insert(\{p_1, p_2, p_3, ..., p_m\})$
                $score\_list.insert(score)$
            end
        end
    end
end
Output: best values of hyperparameters $cv\_list[score\_list.index(max(score\_list))]$
```

4.4. Function-preserving Data Augmentation

One critical challenge of code-reuse exploit detection is sample diversity. One feasible solution to tackle this challenge involves the adoption of data augmentation. In the data augmentation module, we endeavor to generate more samples to artificially increase the variety of input instances for training phase to enhance the generalization of our detection model.

As is known, the performance of the learning-based model is determined by the training data. However, under realistic conditions, researchers are plagued by the limited available data. For example, we can only collect 68 distinct public-available code-reuse exploits with completely disclosed PoC (Proof-of-Concept) in open-source community ExploitDB [48] (accessed on September 15th 2022). The limited publicly available samples constrain the transferability and generalization of trained model on practical application. That means the feature distribution of code-reuse exploits in a real-world environment probably differs from the feature distribution in scarce local training set.

Therefore, to generate more exploits expectantly on the adversarial side can promisingly raise the effectiveness of trained model. Data augmentation can produce more samples to increase the diversity and quantity of training data. It means that the newly generated samples might cover possible polymorphic patterns or new paradigms, which can improve the cognitive ability of trained model.

During data augmentation, we are supposed to guarantee the integrity of functionalities and semantics to avoid infeasible generations. Assuming that we have full knowledge of code-reuse samples in the original dataset, we can generate more samples by subtle manipulations in gadget level while keeping generated samples "real" threats. For a gadget in the original sample, we might find alternative gadgets with the same instructions from optional libraries loaded in memory. For example, we can find 358 alternative gadgets in the common library *msvcrt.dll* to complete *POP EAX; RET* instructions, and 58 alternative gadgets in the common library *msvcr71.dll* to complete *PUSH ESP; RET* instructions. We might use those gadgets with the same instructions to replace the original ones. Also, we can inject redundant gadgets at proper positions of a payload while not altering the functionality of the original gadget chain. As an example, we can inject gadgets which involve operations on registers not later mentioned in the gadget chain. Additionally, we can rephrase the gadgets by using some semantically equivalent gadgets. For instance, we can rephrase the gadget "*0x7C37653D, # POP EAX # POP EDI # POP ESI # POP EBX # POP EBP # RETN*" into several decomposed pieces of short gadgets.

Concretely, for malicious samples, we adopt three main manipulation schemes as follows:

- Synonym Replacement (SR) - Randomly choose the address value of a gadget and replace it with the address value of alternative synonymous gadget.
- Redundant Insertion (RI) - Randomly insert the address values of redundant gadgets at proper positions of the payload.
- Rephrasing Transform (RT) - Rephrase a gadget while retaining instruction semantics, and then replace the original address value with the address value of rephrased gadget.

For benign samples, we just adopt random mutation to mimic the noise data to increase the randomness. The process can be shown schematically in Figure. 8.

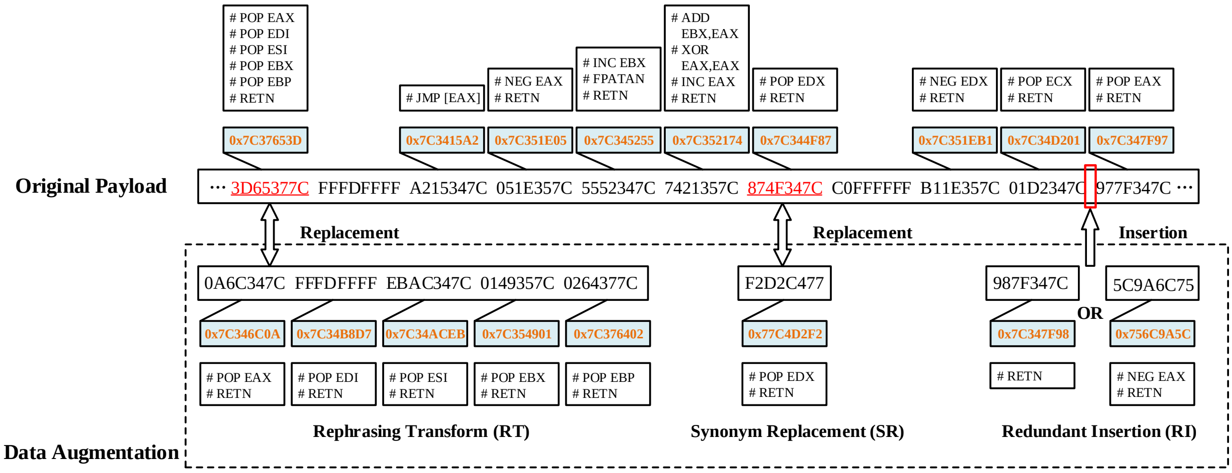


Figure 8. Schematic diagram of proposed function-preserving data augmentation on a malicious sample.

And we can conclude the procedure of data augmentation in the form of pseudocode shown in Algorithm 2:

Algorithm 2: Data augmentation

```
Input: $\mathcal{X} = \{\{(\mathbf{x}_i, y_i) \mid i = 1, \ldots, n\}, \mathbf{x}_i \in \mathbb{R}^p, y_i \in \{-1, +1\}\}_{i=1}^m$, Augmentation
       function set $F_{aug}$
Initialize $\mathcal{X}_{aug}^{(i)} = \mathcal{X}$
for each $f_{aug}^{(i)} \in F_{aug}$ do
    $\mathcal{X}_{aug}^{(i)} \leftarrow f_{aug}^{(i)}(\mathcal{X})$
    $\mathcal{X} \leftarrow \mathcal{X} \bigcup \mathcal{X}_{aug}^{(i)}$
end
Output: Enhanced dataset $\mathcal{X}' = \mathcal{X}$
```

In practical implementation, we build a corpus composed of candidate gadgets chosen from common libraries beforehand, which provides raw materials for our data augmentation.

4.5. Corpus-based Interpreter Module

In the interpreter module, we implement a lightweight interpreter to provide tentative and preliminary dissections for further forensic on suspicious data.

The goal of interpreter is not to conduct further deterministic detection, but to endeavor to present auxiliary information to the analyst for further investigation and to shorten the analytical duration. Considering that current interpreter cannot achieve enough high detection performance as expected, using interpreter as a double checker might actually encumber the overall performance of whole framework. Hence, in our design, we regard the interpreter module as an auxiliary component to support detection decisions.

Unlike previous approaches, we do not require the memory snapshots or activation context to be reproduced to get a comprehensive result. We take full advantage of the fact that our interpretation is performed on the fundamental of detection to make our interpreter as lightweight as possible. We leverage corpus-based mechanism to annotate and highlight underlying "exploit clues" to enhance the persuasiveness of the antecedent detector, and provide speculative interpretation on those suspicious bytes. And those critical "exploit clues" on exploit techniques can provide auxiliary evidence for reliability and interpretability.

We consider those clues in two aspects: invariant gadgets and guessed gadgets.

Under practical circumstance, massive code-reuse payloads contain gadgets that exist in the non-ASLR code segments to reduce exploit difficulty, which remain invariant across different environment configurations. We collect such universal invariant gadgets from frequently-used non-randomized libraries from Windows and Linux platforms, and construct a corpus covering the gathered gadgets. The gadgets inserted into the corpus can be classified into two cases. In the first case, the gadgets are some critical function API such as *VirtualAlloc()*, *VirtualProtect()*, *mmap()*, *HeapAlloc()*, and *SetInfoProcess()*, which plays a significant role in practical application. In the other case, the gadgets are instruction sequences taken from some unrebased DLLs such as *icucnv36.dll*, *mfc71u.dll* and *msvcr71.dll*, which perform some basic operations such as move register, stack pivot, and jump operations. In practice, we examine the data stream by byte and attempt to sort out such fixed and hard-coded address values of gadgets with preferred base address from corpus.

Next, we adopt a guess mechanism to acquire possible gadgets. Modern operating systems randomize only the base address of a code module, leaving the relative distances between the gadgets in a component unchanged. We adopt a guess mechanism stated in Algorithm 3 similar to literature [44] which leverages ASLR implementation details to guess the base address. In fact, Windows only attempts to randomize 8 bits of a 32-bit address, from bits 16 to 23. As a result, in a brute-force situation, we can guess the base address of a binary in 256 trials. We collect common DLLs and use them as corpus for translation. For every guess, we perform tentative corpus-based translation and verify the validity of each guess based on some criteria.

Since the tentative translation is performed based on detected suspicious data, we can relax the criteria of judgment unlike in the literature [44]. We borrowed the ideas from previous work [11,25] to propose the criteria. The *criteria* regulates the determination on a single gadget and a gadget chain. For the determination on a single gadget, we set the maximum instruction length G_size as 5 [11]. Once we locate a possible gadget, we continue to conduct corpus-based translation on subsequent bytes with same base address to find more gadgets. As the literature [25] declared, there do not exist any benign input with more than three unique gadgets. Hence, for the determination on a gadget chain, we set a minimum length C_size as 4. Concisely, when we find four consecutive possible gadgets, we can deduce the base address.

Algorithm 3: Guess Algorithm

```
Input: suspicious data $DS = [Byte_1, Byte_2, Byte_3, ...Byte_L]$, candidate base address
       set $Base_j|_{j=0}^{m}$
for $base\ idx \in L$ do
    $Addr = DS[idx, idx + 3]$
    for $base\ Base_i \in \{Base_j\}|_{j=0}^{m}$ do
        $InstSeq = CorpusTrans(Addr + Base_i)$
        if $InstSeq\ satisfy\ \mathfrak{Criteria}$ then
            $Base = Base_i$
            Return $Base$
        end
    end
end
Output: Guessed base address $Base$
```

Accordingly, we can conduct tentative corpus-based translations on the rest of suspicious data to recover more gadgets, provided that these gadgets come from the DLLs within our corpus. Hence, the effectiveness of our interpreter largely depends on the magnitude of corpus to input samples.

After preliminary translations on those suspicious data, we can acquire some "exploit clues". Identifications of such "exploit clues" can provide auxiliary information to help researchers better understand and analyze code-reuse exploits. And the detection results with such supporting evidence can bridge the gap between suspicion and determinism and enhance the reliability of decisions.

Because a sliding-window decision is made in the detection module, the search space is extensively reduced. Intuitively, our approach does not incur a perceptible performance overhead or cause a substantial impact on the whole system's throughput. Experiments show that our interpreter module only leads to about 1.5% extra time consumption.

5. Evaluation

In this section, we demonstrate a concise and precise description of the experimental configurations and results, their explanations as well as the experimental conclusions that can be drawn. First, we introduce the construction of dataset and experimental settings. Then we illustrate the comparative experiments with some state-of-the-art approaches to validate the effectiveness of our proposed methods. At last, we present some ablation studies for further investigation on the effectiveness and optimization of each component.

5.1. Dataset and Experimental Settings

This subsection describes the construction and composition of dataset, and the detailed configurations of experiments.

5.1.1. Dataset construction

Our experiments were conducted on a dataset which consists of piratical code-reuse samples and part of the USTC-TFC2016 dataset [49]. To simplify and facilitate comparative experiments, we only consider exploit attacks towards program of x86 architecture when building datasets. The samples are stored in the form of PCAP files.

Unlike previous work, we employ code-reuse exploits towards real-world vulnerable Windows and Linux programs instead of hand-crafted or simulated code-reuse payloads. This makes our experiment more convincing. Detailed, we collected code-reuse exploits and PoC(Proof of Concept) scripts from Exploit-DB [48], GitHub, PacketStrom [50], VirrusTotal [51], VirusShare [52], and Corelan ROPdb [53]. All the involved samples and PoC can be publicly available or acquired by request from mentioned websites. Then we generated code-reuse samples based on PoC scripts, and gathered those generated samples with existing exploits to build malicious part of the dataset in our study. The malicious samples are transmitted over different application-layer protocols, such as HTTP, and FTP, etc.

Benign samples in our study are randomly chosen from USTC-TFC2016 dataset. USTC-TFC2016 is a prominent open-source dataset of unencrypted traffic, collected from a real network environment and IXIA BPS simulation equipment. The benign part of USTC-TFC2016 contains packets from ten different applications, listed as: Facetime, Skype, BitTorrent, Gmail, Outlook, WOW(entertainment application WorldOfWarcraft), MySQL, Weibo, FTP(File Transport Protocol message), and SMB(windows Server Massage Block). We randomly sample the packets of each type in the same amount to form the benign part of our dataset.

The details of our dataset are shown in Table 1. Because the number of disclosed code-reuse samples is indeed limited in diversity, the collected and generated samples are still few, even with the best efforts. But there is no redundancy and repetitive code-reuse exploits in the dataset, thus the quality of data can be fully guaranteed. The number of benign and malicious samples is the same, both 10,000. During experiments, the dataset is divided into training and test sets according to the 5: 5 ratio.

Table 1. Details of dataset used in this paper.

Label	Type	Amount
Benign	Factime	1,000
	Skype	1,000
	Bittorent	1,000
	Gmail	1,000
	Outlook	1,000
	WOW	1,000
	MySQL	1,000
	Weibo	1,000
	FTP packets	1,000
	SMB packets	1,000
Malicious	HTTP packets	4,400
	FTP packets	4,000
	Other network packets	1,600

5.1.2. Environment configuration

All experiments were performed under the same experiment configuration, and the details are listed in Table 2.

Table 2. Hardware and software settings in our experiments.

Category	Component	Description
Hardware	Central processing unit	Intel Xeon E5-2678 v3
	Random access memory	DDR4 128 GB
	Graphics processing unit	RTX 3090
Software	Operating system	Ubuntu 18.04 LTS
	Programming language	Python 3.8.11
	Algorithm libary	Sklearn 0.24.2
	Neural network engine	Tensorflow 2.4.0

5.2. Performance Evaluation

In this part, we describe the experiments to verify the effectiveness of our method and discuss the conclusions revealed behind the results.

5.2.1. Performance metrics

We adopted five popular metrics in our experiments to weigh the classification performance. The accuracy(ACC) is defined as the number of correctly predicted samples out of all the samples.

$$\mathrm{ACC} \ = \frac{TP+TN}{TP+TN+FP+FN} \tag{7}$$

The precision(PRE) refers to proportion of malicious identifications is actually correct.

$$\mathrm{PRE} \ = \frac{TP}{TP+FP} \tag{8}$$

The recall(REC) is calculated as the number of malicious samples correctly classified out of malicious samples.

$$\mathrm{REC} \ = \frac{TP}{TP+FN} \tag{9}$$

The false alarms rate(FAR) is the proportion of false alarms compared to all malicious predictions

$$\mathrm{FAR} \ = \frac{FP}{FP+TN} \tag{10}$$

The F1-score(F1) is the harmonic mean of precision and recall.

$$\mathrm{F1} \ = 2 \times \frac{\text{Precision } * \text{ Recall}}{\text{Precision } + \text{ Recall}} \tag{11}$$

where:

- True Positive (TP)— Malicious sample that is correctly classified as an attack.
- False Positive (FP)— Benign sample that is incorrectly classified as an attack.
- True Negative (TN)— Benign sample that is correctly classified as normal.
- False Negative (FN)— Malicious sample that is incorrectly classified as normal.

5.2.2. Baseline methods

We evaluate the performance of our approach by comparing it with several state-of-the-art baseline methods proposed in previous studies. We reproduced those methods to weigh the performance of our method under the same environment. The methods to be compared include:

- Literature [31]: sequential probability ratio test (SPRT) with probabilistic inspection on packets for the code-reuse exploits detection.
- eavesROP [33]: matching filter algorithm with fast fourier transform preprocessing for code-reuse exploits detection.
- ROPNN [35]: lightweight deep neural network methods for the code-reuse exploits detection.
- STROP [38]: static inspection based on seven statistical characteristics of code-reuse payloads to recognize the specialized exploits.
- n-ROPdetector [42]: pattern matching with predefined rule for the code-reuse exploits detection.

5.2.3. Experiment results

To verify the effectiveness of our method, we conducted comparative experiments based on the aforementioned dataset to compare the performance between the baselines with proposed method. The experimental results for each detection performance are reported in Table 2. From the results obtained, the proposed method significantly outperform the baseline methods in all four metrics. Due to the obvious gaps in performance indicators, we can deduce the prominent effectiveness of our method and exclude contingent effects of experiments.

Table 3. Effectiveness comparison with baselines.

Method	ACC	PRE	REC	F1	FAR
Literature [31]	0.8702	0.9012	0.8316	0.8650	0.0912
eavesROP	0.9320	0.9553	0.9064	0.9302	0.0424
ROPNN	0.9410	0.9022	0.9892	0.9437	0.1072
STROP	0.8354	0.9584	0.7012	0.8099	0.0304
n-ROPdetector	0.8108	0.9346	0.6684	0.7794	0.0468
Our method	**0.9838**	**0.9779**	**0.9900**	**0.9839**	**0.0224**

Subsequently, we compare the efficiency of our method with the baselines. We utilized professional test equipment named *SpirentTestCenter* to generate different types of traffic for measuring the throughput of each approach. During the test, we set the ratio of random generated normal traffic to code-reuse exploit traffic as 100:1 to simulate network traffic like in a real-world deployment environment. The test was lasting for 10 hours, and all the approaches are executed under same configuration. We calculate the total size of packets passing through those systems, and list the throughout in Table 4. It can be observed from this table that the throughput of our method is pretty competitive which can satisfy the requirements of practical applications. The rule-based engine makes *n-ROPdetector* work more efficiently than other methods. The efficiencies of eavesROP, STROP, and literature [31] are restricted by cumbersome calculations. Due to the usage of lightweight model architecture, the efficiency of *ROPNN* is unexpectedly high, especially under GPU computation. Although the throughput of our method is slightly lower than *n-ROPdetector* and *ROPNN*(working on GPU mode), the efficiency is acceptable considering the detection performance.

Table 4. Efficiency comparison with baselines.

Method	Throughput($Mbit \backslash sec$)
Literature [31]	380.50
eavesROP	93.20
ROPNN	552.12(with GPU), 497.05(without GPU)
STROP	82.04
n-ROPdetector	570.01
Our method	524.32

5.2.4. Effectiveness of interpreter

To further demonstrate the necessity and effectiveness of interpreter module in *Horus* framework, we perform statistical analysis and the details are shown in Table 5.

Table 5. Performance of interpreter.

Type	Samples Interpreted	Gadgets Interpreted	Gadgets Correctly Interpreted	Samples with Helpful Interpreted
Portion	0.8917	0.6728	0.9630	0.8113

In our experiment, the detection module presented 2,531 suspicious samples with precisely windowed bytes, including 2,475 true positives and 56 false positives. The interpreter module works on the suspicious bytes to produce interpretations. Of true positive samples, our interpreter module successfully generated interpretations on 2,207 samples, namely a portion of 89.17%. Since the interpretation is preliminary and partial, over all the gadgets, only 67.28% were tentatively interpreted. And 96.30% of them were correctly interpreted. One important reason of misinterpretation is that those code-reuse exploits extensively rely on some private binaries or third-party libraries, which are out of the scope of our corpus. In general, the exploits are promising to be interpreted correctly if it is crafted based on common libraries. Overall, our interpreter module provided corrected interpretation on 2,008 samples out of all true positives, which proves the validity. And the average overhead of the interpreter module is 1.5% based on massive experiments, which indicated the efficiency of our interpreter module.

5.3. Ablation Studies

We conducted ablation studies to understand the effectiveness of several core techniques in our approach and the influence from properties of code-reuse exploits. In particular, our experiments mainly focus on answering the five research questions:

- **RQ1**: Impact of applying time-series or frequency-domain features alone on detection performance and whether the fusion of time-series and frequency-domain features can achieve better performance.
- **RQ2**: Impact of applying different backend classification algorithms and why we choose LightGBM as core classification algorithm.
- **RQ3**: Impact of applying data augmentation on detection performance and the separate effectiveness of different augmentation strategies.
- **RQ4**: Impact of different sampling window size, and the optimal selection in our experiment.
- **RQ5**: Detection capability of our method, and the impact of payload length on detection performance.

5.3.1. RQ1: Time-series features versus frequency-domain features

To evaluate the effect of time-series and frequency-domain features on detection performance, we trained our model in different feature fields. Then we repeat the test phase and list the performance identifiers in Table 6.

It is demonstrated that the model on collaborative time-frequency features can bring better performance. And our results can provide sufficient proofs to confirm the complementarity of time-series and frequency-domain features.

Table 6. Comparisons of different feature selection strategies.

Feature	ACC	PRE	REC	F1	FAR
Time-series	0.9522	0.9333	0.9740	0.9532	0.0696
Frequency-domain	0.9628	0.9699	0.9552	0.9625	0.0296
Both	**0.9838**	**0.9779**	**0.9900**	**0.9839**	**0.0224**

5.3.2. RQ2: Impact of classifier selection

In order to gain more insight into the impacts of classifier model, we performed the experiments with different backend model, and the results are shown in Table 7. Compared to Navie Bayes, SVM, and Decision Tree, ensemble models like lightGBM integrate the output of multiple weak classifiers, which can significantly increase the performance. Compared to Random Forest and XGBoost, lightGBM produces much more complex trees to achieve higher accuracy. Due to the usage of dropout, the performance of DART is slightly weaker than GBDT. Above all, the results indicate that lightGBM (GBDT) is the best model to our needs.

Table 7. Comparisons of different backend classifier algorithms.

Method	ACC	PRE	REC	F1	FAR
Naive Bayes	0.9414	0.9434	0.9392	0.9413	0.0564
SVM	0.9602	0.9611	0.9592	0.9602	0.0388
Decision Tree	0.9580	0.9558	0.9604	0.9581	0.0444
Random Forest	0.9796	0.9720	0.9876	0.9798	0.0284
XGBoost	0.9816	0.9770	0.9864	0.9817	0.0232
lightgbm(DART)	0.9822	0.9759	0.9888	0.9823	0.0244
lightgbm(GBDT)	**0.9838**	**0.9779**	**0.9900**	**0.9839**	**0.0224**

5.3.3. RQ3: Impact of data augmentation

To enable a performance comparison between different data augmentation operations and verify the effectiveness of data augmentation, we conducted extensive experiments and recorded the respective results as in Table 8.

Table 8. Comparisons of different data augmentation(DA) operations.

Augmentation strategies	ACC	PRE	REC	F1	FAR
Without DA	0.9408	0.9498	0.9308	0.9402	0.0492
SR only	0.9552	0.9611	0.9488	0.9549	0.0384
RI only	0.9468	0.9555	0.9372	0.9463	0.0436
RT only	0.9668	0.9683	0.9652	0.9667	0.0316
SR+RI	0.9624	0.9650	0.9596	0.9623	0.0348
SR+RT	0.9778	0.9731	0.9828	0.9779	0.0272
RI+RT	0.9718	0.9712	0.9724	0.9718	0.0288
SR+RI+RT	**0.9838**	**0.9779**	**0.9900**	**0.9839**	**0.0224**

As can be seen in Table 8, out of the three operations, Rephrasing Transform(RT) can provide the largest improvement. It means that Rephrasing Transform(RT) can provide bigger discrepancies from original samples to enlarge the diversity. And if we combine multiple operations, the performance will rise due to the production of more complex variants. When combining all three operations, we get the best performance as expected. Therefore, in our framework, we combine all the three operations during training.

5.3.4. RQ4: Window size selection

One of the primary influence factor on the detection performance is the size of sampling window. In this part, we explore the influence of different sizes to obtain the optimal window size. We focus on the varying curve of two identical indicators: ACC and FAR. And we recorded the ACC and FAR under different window sizes and plot two curves to depict the trends in Figure 9.

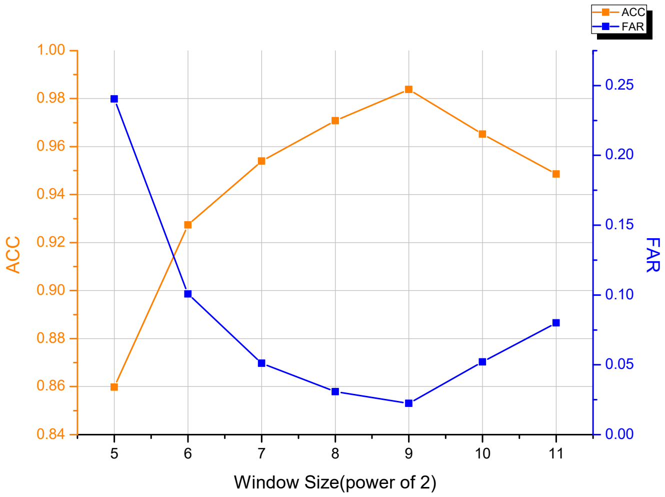


Figure 9. Performance on different window sizes.

Considering the memory organization of modern computer system, we set the window sizes in power of 2. It can be observed from Figure 9 that DR increases gradually from 32 to 512, meanwhile FAR decreases monotonically. When the window size exceeds 512, DR and FAR deteriorate along with the continual increase of window size. The results show that the best window size is 512 to reach the best DR and FAR under our experimental configuration. The curves indicate that when the window size is under a certain threshold, smaller windows are prone to be influenced by local noises or truncations. And when the window size surpasses a certain threshold, the ACC and FAR also decrease because of the degradation of signal-to -noise ratio along with the increase of window size.

5.3.5. RQ5: Detection capability

To better understand the detection capability of *Horus*, we take a closer inspection on the performance of *Horus* on different samples. Although there are many factors that affect detection performance, a more general measurement is payload length. Concretely, we take an exploration on the influence of payload length on detection performance of *Horus*. We focus on the Recall(or namely detection rate) of different length ranges of code-reuse payloads. Within the experiments, all other configurations are set by default. Recalls on different payload length are depicted in Figure 10.

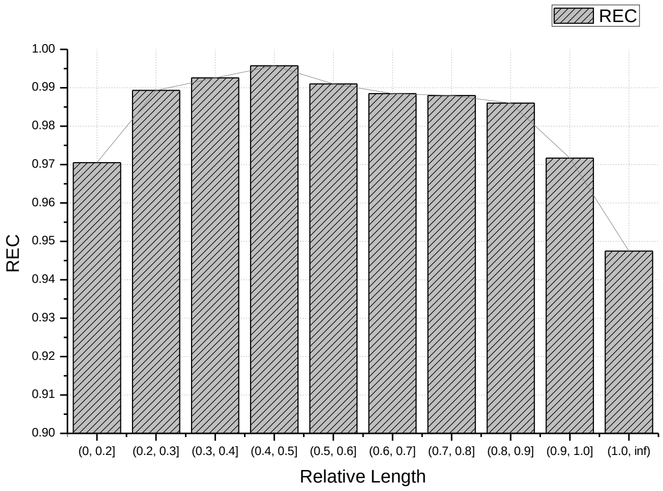


Figure 10. Detection capability on payloads of different lengths.

In the experiments, we utilize the relative length (ratio to window) to express the payload length. According to the relative length, we split the lengths into ten groups with respective bars. The results in Figure 10 reveal that when handling samples with a payload of 40-50% window size, we can obtain the best detection performance. When we deal with samples with short payloads, the background noise will bring a significant negative influence. On the other hand, once the relative length increase by over 50%, it means that truncation occurs. Although the noise bytes are largely eliminated, truncation also damages the integrity of the structure of a code-reuse payload and causes loss of some critical features. Under actual conditions, the relative lengths of the payload are an important factor that affects the detection capability of our *Horus* framework.

6. Discussion

In this section, we first summarize the experimental results. Then we clarify the strengths and limitations in our proposed *Horus* which indicate possible future research directions.

The presented study confirmed the findings about how to enhance the effectiveness and reliability of code-reuse exploits detection. Due to the lack of publicly available data, we take efforts to collect real-world samples to build a comprehensive dataset shown in Table 1. Then we conduct comparative experiments with some baseline approaches under constructed dataset and the same experiment(see Table 2). The prominent detection performance is elaborately clarified in Table 3 to support the effectiveness of our design. Meanwhile, through stimulated experiments, we demonstrate the competitive throughput of our framework in Table 4. Next, we calculate the accuracy of proposed corpus-based interpreter. As shown in Table 5, of the gadgets can be correctly translated, and of samples can be generated interpretation. Ablation studies are further presented to provide exploration on influencing factors of performance and emphasize the effectiveness of model design and data augmentation strategies. Based on the experiments and analysis, the effectiveness and reliability can be proved. Moreover, we can clarify it a promisingly realistic solution can be integrated into real-world defense systems.

Compared to previous solutions, we have a deeper investigation on the value characteristics of code-reuse exploits. *Horus* works on the data stream irrespective of the target program. In our framework, not only the detection results will be output, but also the corresponding explanations used as supporting information. The feature of "decision with interpretation" helps to mitigate the influence of a black-box problem widely existed in learning-based systems, work as a trail to bridge the gap between statistical and deterministic strategies. Moreover, the application of data augmentation help to solve the diversity problem in current methods, and further raise the effectiveness of *Horus*, especially on unseen patterns. In addition, *Horus* can be easily deployed on the network edge or applied as NIDS without the requirement of dynamic execution.

However, *Horus* still has some limitations, which designates the direction of future improvement. One basic assumption of our framework is that the code-reuse payloads remain visible when transmitted over steaming data. That means the dynamically generated code-reuse payloads and obfuscated or encrypted payloads cannot be recognized by *Horus*. Especially, the encryption problem has been a main handicap for nowadays deep packet inspection systems. To tackle this challenge, an extra module like *BlindBox* [55] or *BlindIDS* [56] might be employed for frontend process to recover encrypted traffic to plain text. Notice that our framework uses value characteristic of consecutive bytes as the theoretic basis. When encountered with exploits using extremely short gadget chains, *Horus* might be invalidated. Also, the size of sampling window is still an inevitable influencing factor. As discussed in RQ4 and RQ5, the selection of window sizes is a trade-off problem under different conditions.

7. Conclusions

In this paper, we develop *Horus*, a real-time framework that enables effective and reliable detection on code-reuse exploits hidden in data stream. We take a deep investigation on the value characteristics of code-reuse exploits to carry out the basic idea of the *Horus* framework. Unlike traditional defense systems, *Horus* focuses on end-to-end detection (as a service) rather than built-in prevention on end-user systems. *Horus* statically examines every packet transmitted over network traffic without any prior knowledge, output suspicious packets with tentative interpretations based on corpus-based interpreter. Concretely, *Horus* utilizes collaborative analysis with data augmentation on both time series and frequency domain to enhance the detection performance. Experiments show that *Horus* can achieve an outstanding performance with a competitive throughput for real-world application. *Horus* not only reveals the presence of a code-reuse payload, but also recovers the possible attack semantics as auxiliary for further forensics. Our research provides a promising design to shed light on the interpretability and verifiability dilemma of a data-driven method. Due to the portability of *Horus*, it can work on the network edge alone or be integrated into an existing defense system.

Promising extensions of the present study are divided into three major aspects. We first plan to employ an adaptive window in order to accelerate the sampling process. By using an adaptive window to substitute for the fix-length window, we can reduce the sampling times and help to better focus on suspicious data segments. Secondly, we intend to enrich the data augmentation strategies for further extend the diversity of training data. In *Horus*, we adopt some mutation-based operations and explore the performance, respectively. We will incorporate generation-based methods to produce more candidate data. Besides, we will continually explore feasible strategies to extend the capacity of *Horus* to adapt to the obfuscated or encrypted payloads.

Author Contributions: Conceptualization, G.Y. and X.L.; methodology, G.Y.; software, G.Y.; validation, G.Y. and X.L.; formal analysis, X.L.; investigation, C.T.; resources, C.T.; data curation, G.Y.; writing—original draft preparation, G.Y.; writing—review and editing, X.L. and C.T.; visualization, G.Y.; supervision, C.T.; project administration, C.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable

Informed Consent Statement: Not applicable

Data Availability Statement: The data presented in this study are available on request from the corresponding author after publication. The data are not publicly available due to privacy or ethical restrictions.

Acknowledgments: Thanks to INSG Lab of NUDT for providing high-performance server to support our research.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

NIDS	Network-based Intrusion Detection System
ASLR	Address Space Layout Randomization
DEP	Data Execution Prevention
DLL	Dynamic Link Library
ROP	Return oriented programming
JOP	Jump oriented programming
PoC	Proof of Concept
ACC	Accuracy
PRE	Precision
REC	Recall
F1	F1-score
FAR	False alarm rate
TP	Ture positive
FP	False positive
TN	Ture negative
FN	False negative

References

1. Wojtczuk, R. The advanced return-into-lib (c) exploits: Pax case study. *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e* **2001**, *70*.
2. Shacham, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Proceedings of the Proceedings of the 14th ACM conference on Computer and communications security, 2007, pp. 552–561.
3. Bletsch, T.; Jiang, X.; Freeh, V.W.; Liang, Z. Jump-oriented programming: a new class of code-reuse attack. In Proceedings of the Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, 2011, pp. 30–40.
4. Ispoglou, K.K.; Albassam, B.; Jaeger, T.; Payer, M. Block Oriented Programming: Automating Data-Only Attacks. In Proceedings of the the 2018 ACM SIGSAC Conference, 2018.
5. Carlini, N.; Wagner, D. {ROP} is still dangerous: Breaking modern defenses. In Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14), 2014, pp. 385–399.
6. Szekeres, L.; Payer, M.; Wei, T.; Song, D. Sok: Eternal war in memory. In Proceedings of the 2013 IEEE Symposium on Security and Privacy. IEEE, 2013, pp. 48–62.
7. Chen, P.; Xiao, H.; Shen, X.; Yin, X.; Mao, B.; Xie, L. DROP: Detecting return-oriented programming malicious code. In Proceedings of the International Conference on Information Systems Security. Springer, 2009, pp. 163–177.
8. Davi, L.; Sadeghi, A.R.; Winandy, M. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In Proceedings of the Proceedings of the 2009 ACM workshop on Scalable trusted computing, 2009, pp. 49–54.
9. Pappas, V.; Polychronakis, M.; Keromytis, A.D. Transparent {ROP} exploit mitigation using indirect branch tracing. In Proceedings of the 22nd USENIX Security Symposium (USENIX Security 13), 2013, pp. 447–462.
10. Cheng, Y.; Zhou, Z.; Miao, Y.; Ding, X.; Deng, R.H. ROPecker: A generic and practical approach for defending against ROP attack **2014**.
11. Nie, M.; Su, P.; Li, Q.; Wang, Z.; Ying, L.; Hu, J.; Feng, D. Xede: Practical exploit early detection. In Proceedings of the International Symposium on Recent Advances in Intrusion Detection. Springer, 2015, pp. 198–221.
12. Davi, L.; Sadeghi, A.R.; Winandy, M. ROPdefender: A detection tool to defend against return-oriented programming attacks. In Proceedings of the Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, 2011, pp. 40–51.
13. Si, L.; Yu, J.; Luo, L.; Ma, J.; Wu, Q.; Li, S. ROP-Hunt: Detecting Return-Oriented Programming Attacks in Applications. In Proceedings of the International Conference on Security, Privacy and Anonymity in Computation, Communication and Storage. Springer, 2016, pp. 131–144.
14. Portokalidis, G.; Slowinska, A.; Bos, H. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *ACM SIGOPS Operating Systems Review* **2006**, *40*, 15–27.
15. Shioji, E.; Kawakoya, Y.; Iwamura, M.; Hariu, T. Code shredding: byte-granular randomization of program layout for detecting code-reuse attacks. In Proceedings of the Proceedings of the 28th annual computer security applications conference, 2012, pp. 309–318.
16. Xia, Y.; Liu, Y.; Chen, H.; Zang, B. CFIMon: Detecting violation of control flow integrity using performance counters. In Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012). IEEE, 2012, pp. 1–12.
17. Pappas, V. kBouncer: Efficient and transparent ROP mitigation. *Apr* **2012**, *1*, 1–2.
18. Tymburibá, M.; Emilio, R.; Pereira, F. Riprop: A dynamic detector of rop attacks. In Proceedings of the Proceedings of the 2015 Brazilian Congress on Software: Theory and Practice, 2015, p. 2.
19. Jacobson, E.R.; Bernat, A.R.; Williams, W.R.; Miller, B.P. Detecting code reuse attacks with a model of conformant program execution. In Proceedings of the International Symposium on Engineering Secure Software and Systems. Springer, 2014, pp. 1–18.

20. Kayaalp, M.; Schmitt, T.; Nomani, J.; Ponomarev, D.; Abu-Ghazaleh, N. SCRAP: Architecture for signature-based protection from code reuse attacks. In Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2013, pp. 258–269.
21. Elsabagh, M.; Barbara, D.; Fleck, D.; Stavrou, A. Detecting ROP with statistical learning of program characteristics. In Proceedings of the Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, 2017, pp. 219–226.
22. Das, S.; Chen, B.; Chandramohan, M.; Liu, Y.; Zhang, W. ROPSentry: Runtime defense against ROP attacks using hardware performance counters. *Computers & Security* **2018**, *73*, 374–388.
23. Stancill, B.; Snow, K.Z.; Otterness, N.; Monrose, F.; Davi, L.; Sadeghi, A.R. Check my profile: Leveraging static analysis for fast and accurate detection of rop gadgets. In Proceedings of the International Workshop on Recent Advances in Intrusion Detection. Springer, 2013, pp. 62–81.
24. Ying, Y. UnROP: creating correct backtrace from core dumps with stack pivoting. PhD thesis, University of Georgia, 2014.
25. Polychronakis, M.; Keromytis, A.D. ROP payload detection using speculative code execution. In Proceedings of the 2011 6th International Conference on Malicious and Unwanted Software. IEEE, 2011, pp. 58–65.
26. Lu, K.; Zou, D.; Wen, W.; Gao, D. deRop: removing return-oriented programming from malware. In Proceedings of the Proceedings of the 27th Annual Computer Security Applications Conference, 2011, pp. 363–372.
27. Graziano, M.; Balzarotti, D.; Zidouemba, A. ROPMEMU: A framework for the analysis of complex code-reuse attacks. In Proceedings of the Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, 2016, pp. 47–58.
28. Zhao, L.; Wang, R.; Wang, L.; Cheng, Y. Reversing and identifying overwritten data structures for memory-corruption exploit diagnosis. In Proceedings of the 2015 IEEE 39th Annual Computer Software and Applications Conference. IEEE, 2015, Vol. 2, pp. 434–443.
29. Göktaş, E.; Athanasopoulos, E.; Polychronakis, M.; Bos, H.; Portokalidis, G. Size Does Matter: Why Using {Gadget-Chain} Length to Prevent {Code-Reuse} Attacks is Hard. In Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14), 2014, pp. 417–432.
30. Ho, J.W. Code-reuse attack detection using Kullback-Leibler divergence in IoT. *International journal of advanced smart convergence* **2016**, *5*, 54–56.
31. Ho, J.W. Efficient and robust detection of code-reuse attacks through probabilistic packet inspection in industrial IoT devices. *IEEE Access* **2018**, *6*, 54343–54354.
32. Ho, J.W.; Kang, S.; Kim, S. Poster: Code-Reuse Attack Detection Using Sequential Hypothesis Testing in IoT, 2017.
33. Jämthagen, C.; Karlsson, L.; Stankovski, P.; Hell, M. eavesrop: Listening for rop payloads in data streams. In Proceedings of the International Conference on Information Security. Springer, 2014, pp. 413–424.
34. Li, X.; Hu, Z.; Wang, H.; Fu, Y.; Chen, P.; Zhu, M.; Liu, P. DeepReturn: A deep neural network can learn how to detect previously-unseen ROP payloads without using any heuristics. *Journal of Computer Security* **2020**, *28*, 499–523.
35. Li, X.; Hu, Z.; Fu, Y.; Chen, P.; Zhu, M.; Liu, P. ROPNN: Detection of ROP payloads using deep neural networks. *arXiv preprint arXiv:1807.11110* **2018**.
36. Wang, H.; Liu, P. Tackling Imbalanced Data in Cybersecurity with Transfer Learning: A Case with ROP Payload Detection. *arXiv preprint arXiv:2105.02996* **2021**.
37. Usui, T.; Ikuse, T.; Otsuki, Y.; Kawakoya, Y.; Iwamura, M.; Miyoshi, J.; Matsuura, K. Ropminer: Learning-based static detection of rop chain considering linkability of rop gadgets. *IEICE Transactions on Information and Systems* **2020**, *103*, 1476–1492.
38. Choi, Y.; Lee, D. Strop: Static approach for detection of return-oriented programming attack in network. *IEICE Transactions on Communications* **2015**, *98*, 242–251.
39. Usui, T.; Ikuse, T.; Iwamura, M.; Yada, T. POSTER: Static ROP chain detection based on hidden Markov model considering ROP chain integrity. In Proceedings of the Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 1808–1810.
40. Hsu, F.H.; Guo, F.; Chiueh, T.c. Scalable network-based buffer overflow attack detection. In Proceedings of the Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems, 2006, pp. 163–172.
41. Toth, T.; Kruegel, C. Accurate buffer overflow detection via abstract pay load execution. In Proceedings of the International Workshop on Recent Advances in Intrusion Detection. Springer, 2002, pp. 274–291.
42. Tanaka, Y.; Goto, A. n-ropdetector: Proposal of a method to detect the rop attack code on the network. In Proceedings of the Proceedings of the 2014 Workshop on Cyber Security Analytics, Intelligence and Automation, 2014, pp. 33–36.
43. OConnor, T.; Enck, W. Code-Stop: Code-Reuse Prevention By Context-Aware Traffic Proxying.
44. D'Elia, D.C.; Coppa, E.; Salvati, A.; Demetrescu, C. Static analysis of ROP code. In Proceedings of the Proceedings of the 12th European Workshop on Systems Security, 2019, pp. 1–6.
45. Ye, L.; Keogh, E. Time series shapelets: a new primitive for data mining. In Proceedings of the Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining, 2009, pp. 947–956.
46. Chen, C.; Hsiao, C. Haar wavelet method for solving lumped and distributed-parameter systems. *IEE Proceedings-Control Theory and Applications* **1997**, *144*, 87–94.
47. Ke, G.; Meng, Q.; Finley, T.; Wang, T.; Chen, W.; Ma, W.; Ye, Q.; Liu, T.Y. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems* **2017**, *30*.

48. Exploit Database. EXPLOIT DATABASE BY OFFENSIVE SECURITY. Available online: https://www.exploit-db.com/ (accessed on 15th September 2022).
49. Wang, B.; Su, Y.; Zhang, M.; Nie, J. A Deep Hierarchical Network for Packet-Level Malicious Traffic Detection. *IEEE Access* **2020**, *8*, 201728–201740.
50. Packet Storm. Global Security Resource. Available online: https://packetstormsecurity.com (accessed on 22 Apri 2022).
51. VirusTotal. VirusTotal.com. Available online: https://www.virustotal.com/ (accessed on 22 Apri 2022).
52. VirusShare. VirusShare.com. Available online: https://virusshare.com (accessed on 22 Apri 2022).
53. corelan. Corelan Cybersecurity Research. Available online: https://www.corelan.be/ (accessed on 22 Apri 2022).
54. Abbasi, A.; Wetzels, J.; Bokslag, W.; Zambon, E.; Etalle, S. On emulation-based network intrusion detection systems. In Proceedings of the International Workshop on Recent Advances in Intrusion Detection. Springer, 2014, pp. 384–404.
55. Sherry, J.; Lan, C.; Popa, R.A.; Ratnasamy, S. Blindbox: Deep packet inspection over encrypted traffic. In Proceedings of the Proceedings of the 2015 ACM conference on special interest group on data communication, 2015, pp. 213–226.
56. Canard, S.; Diop, A.; Kheir, N.; Paindavoine, M.; Sabt, M. BlindIDS: Market-compliant and privacy-friendly intrusion detection system over encrypted traffic. In Proceedings of the Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, 2017, pp. 561–574.