APIASO: A Novel API Call Obfuscation Technique Based on Address Space Obscurity

Yang Li, Fei Kang *, Hui Shu, Xiaobing Xiong, Yuntian Zhao and Rongbo Sun

Abstract: API calls are programming interfaces used by applications. When it is difficult for an analyst to perform a direct reverse analysis of a program, the API provides an important basis for analyzing the behavior and functionality of the program. API address spaces are essential for analysts to identify API call information, and therefore API call obfuscation is used as a protection strategy to prevent analysts from obtaining call information from API address spaces. API call obfuscation avoids direct API calls and aims to create a more complex API calling process. Unfortunately, current API call obfuscation methods are not effective in preventing analysts from obtaining usable information from the API address space. To solve this issue, in this paper, we propose an API call obfuscation model based on address space obscurity. The key functions within the API are encrypted and moved to the user code space for execution. This breaks the relationship between the API and its address space, making it impossible for analysts to obtain address information about a known API from the API address space. In our experiments, we developed an archetypical compiler-level API call obfuscation system to automate the obfuscation of input source code into an obfuscated file. The results show that our approach can thwart existing API deobfuscation techniques and is highly resistant to various open-source dynamic analysis platforms. Compared to other obfuscation techniques, our scheme improves API address space obscurity by more than two times, the detection rate of deobfuscation techniques such as Scylla, etc. is zero, and the increase in obfuscation overhead is not more than 20%. The above results show that APIASO has better obfuscation effect and practicability.

Keywords: API address space obscurity; API call obfuscation; anti-reverse analysis; hash function generators

1. Introduction

Windows applications rely on APIs to interact with the system and perform various functions, including network communication, file access, and message interaction. Understanding these APIs is crucial for comprehending program behavior. When direct reverse analysis is challenging, APIs serve as an indispensable reference for analysts to identify program behavior and intent [1].

To protect the API calling process, program developers often employ various obfuscation techniques to prevent detection. The basic premise of API call obfuscation is to use a more complex approach than the Windows API call process to make it difficult for analysts to identify which API has been called. However, over the past two decades, analysts have proposed a range of techniques to overcome API obfuscation. The key step in API deobfuscation is to associate the virtual memory address accessed by the program during runtime with the API name. Analysts typically employ program analysis methods such as symbolic execution or taint analysis to track program execution. This involves collecting call instructions and function addresses, and then correlating them with the loaded APIs in memory to identify any obfuscated APIs.
The ability to collect address information related to known APIs during API calls is a critical factor in the ongoing game between obfuscation and deobfuscation techniques. Sections 2.1 and 2.2 of this paper systematically summarize existing API call obfuscation and deobfuscation techniques. The findings reveal that even the most advanced deobfuscation methods are still capable of collecting addresses associated with known APIs. As a result, none of the currently available API call obfuscation methods can effectively prevent analysts from conducting API deobfuscation. Therefore, the development of a more secure API call obfuscation scheme is a pressing concern.

In this paper, we propose a new approach to API call obfuscation called API address space obscurity (APIASO), which aims to prevent analysts from accessing address information associated with known APIs during the API call process. APIASO protects the entire API call process by moving the API’s critical internal functions into the user code space. When a program calls the API, it first executes the functions that have been moved into the user code space and then proceeds to execute the non-critical functions within the API. This prevents analysts from obtaining information directly related to the known API from the API call. Additionally, for the API code extraction process, our approach provides a wide range of alternative API name hash schemes by designing a hash function generator to achieve more secure API address resolution. The API name hash serves as a cryptographic key to move the key functions of the API to the user code space, making it impossible for analysts to access usable information from the process of API address resolution and function movement.

We conducted extensive experiments to evaluate the effectiveness and obfuscation overhead of APIASO. To quantitatively assess the impact of APIASO on the API address space, we measured the obscurity degree of the API address space. We also tested the resistance of APIASO against four representative deobfuscation techniques. In addition, we demonstrated the availability and correctness of APIASO by developing an automated API call obfuscation system.

The main contributions of this paper are:

- We provide a comprehensive overview of existing API call obfuscation and deobfuscation techniques. We also discuss the limitations of current API obfuscation techniques and their inability to effectively counter advanced deobfuscation techniques.
- We present API address space obscurity (APIASO), an API call obfuscation technique specifically designed for Windows applications. APIASO provides stronger protection than existing API call obfuscation techniques by protecting the entire API call process and preventing analysts from accessing address information associated with known APIs during the API call process.
- We compare APIASO with several other existing API call obfuscation techniques. Our experiments show that APIASO is highly effective in thwarting existing deobfuscation techniques, and it provides a significant increase in the protection strength of program API information.
- We implement an automatic API call obfuscation system based on LLVM [2], which can automatically obfuscate the input program source code. The source code is available on GitHub (https://github.com/Rookiellvm/APIASO, accessed on 13 August 2022).

2. Background

In this section, we systematically provide a review of current API call obfuscation and deobfuscation techniques. While API call obfuscation techniques are commonly utilized for API call protection, analysts have developed numerous deobfuscation techniques to improve analysis efficiency. These techniques employ program analysis techniques such as monitoring program execution for jump instructions and function addresses to expose hidden traces of API calls.

We provide a comprehensive summary of existing mainstream API call obfuscation and deobfuscation techniques to highlight the shortcomings of current API call obfuscation techniques.
techniques when it comes to effectively countering deobfuscation techniques. Finally, we introduce the design goals of APIsASO as a means of addressing these limitations.

2.1. API Call Obfuscation Techniques

Kawakoya [3] et al. provided a formal definition of the concept of API call obfuscation and introduced various specific patterns of API call obfuscation. In a recent study by API-Xray [4], it was revealed that the primary aim of API call obfuscation is to evade the standard API resolution and calling process provided by Windows. Table 1 shows three API call obfuscation methods, each of which is described below.

Table 1. Classification of API call obfuscation techniques.

<table>
<thead>
<tr>
<th>Classification</th>
<th>Pathways</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAT redirection [4]</td>
<td>Anti-debugging, exception triggering, ROP</td>
</tr>
<tr>
<td>Position obfuscation [5]</td>
<td>Instruction stolen, function stolen, DLL stolen</td>
</tr>
<tr>
<td>Call site tampering [6]</td>
<td>GetModuleHandle/LoadLibrary and GetProcAddress</td>
</tr>
</tbody>
</table>

IAT redirection: IAT redirection is achieved by tampering with the IAT table; Figure 1b shows the process of an API call after IAT redirection obfuscation. The API address stored in the IAT table entry is replaced with an “induction area” address. Thus, the address obtained in the IAT does not point directly to the API. Figure 1b shows the redirection method of triggering an exception in the “induction area” by a divide-by-zero operation. In addition, it is possible to redirect API calls by adding anti-debugging functions or ROP techniques to the “induction area”.

![Figure 1. Overview of API call obfuscation techniques.](image-url)

API position obfuscation: Position obfuscation makes the API call address fail to point to the API entry point by moving the API code to execute in user space. Depending on the scope of the moved code, position obfuscation is divided into three cases: (1) Instruction stolen, which moves part of the API instruction to the user code space for execution, and then jumps to the target API code. (2) Function stolen, which moves the entire API to the user code space for execution. Figure 1c describes the process of calling the API after the function is stolen, replacing the API address stored in the IAT table entry with the “induction area” address, which stores the API code copied from the system load DLL space. When the program calls the API, it executes the API in the “induction area”. (3) Dynamic...
Link Library (DLL) stolen, which loads the entire DLL into user code space and calls the API through the self-loaded DLL.

API call site tampering: API call site tampering eliminates the dependence of API calls on IAT and resolves the API address at program runtime. Figure 1d shows the API call process of the program after call site tampering. The program encrypted stores the called API name. When the program calls the API, it decrypts the name and obtains the address of the API through the combination of the functions GetModuleHandle/LoadLibrary and CreateProcAddress. Additionally, to further complicate the analysis, the API name resolution process does not use the aforementioned function calls to obtain the API address. Instead, the program uses the PEB to obtain the DLL address and retrieves the required API address from the export table of the DLL.

2.2. API Deobfuscation Techniques

To improve the efficiency of an analysis, analysts perform API monitoring to obtain API call information for programs that implement API call obfuscation. Table 2 provides an overview of current API deobfuscation techniques. These techniques can be categorized into the following three groups based on their starting point: call site monitoring, position monitoring, and hybrid monitoring.

Table 2. Classification of API deobfuscation techniques.

<table>
<thead>
<tr>
<th>Classification</th>
<th>Citations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call site monitoring</td>
<td>BinUnpack [7], SOK [8], Scylla [9], Eureka [10], RePEc [11], PinDemonium [12], Arancino [13], Arg Prediction [14]</td>
</tr>
<tr>
<td>Position monitoring</td>
<td>API Chaser [15], QuietRIATT [16], Secure unpack [17], Taint-assisted [18]</td>
</tr>
<tr>
<td>Hybrid monitoring</td>
<td>API-Xray [4], RePEconstruct [19]</td>
</tr>
</tbody>
</table>

Call site monitoring: Figure 2 depicts that the deobfuscation techniques for API call site monitoring follow two steps: (1) Instruction scanning (I in Figure 2), which runs PE files to find possible API call sites in memory, including indirect calls, direct calls, or indirect jumps. (2) Address association (II in Figure 2), which correlates the destination address of a possible call site with the exported API address of the loaded dynamic link library. The basic principle of call site monitoring is that the destination address of a jump instruction in memory is always found to be the address of the system-loaded API. As illustrated in Figure 1b,d, there are instructions in the memory space “induction area” in which the target address is the system load API address.

Position monitoring: Position monitoring is the monitoring of system load DLL pages. The basic principle is that regardless of the API call obfuscation technique used, the program eventually executes the system load DLL area code. Position monitoring includes API hook monitoring and taint analysis association. API hook monitoring (III in Figure 2) monitors the execution of API code in the DLL. QuietRIATT and secure unpack use the API hook approach to set hooks in the loaded API code area, which is logged when the program calls the API where the hooks are set; alternatively, the taint analysis association [17,18] (IV in Figure 2) method involves attaching taint tags to the API code, and when the program executes the code in the DLL space, the API executed is determined by the attached taint tags.

Hybrid monitoring: The call site monitoring technique can monitor the call site of the whole program, but the call site address of the program applying API position obfuscation is a non-API-related address, which makes the call site monitoring invalid. The position monitoring approach can effectively prevent API position obfuscation, but it can only address a single path at a time and has low coverage. API-Xray and RePEconstruct have
taken the benefits of both call site monitoring and position monitoring methods into account. They propose a hybrid monitoring approach that combines both techniques to enhance the efficiency of deobfuscation.

**Figure 2.** Overview of API deobfuscation techniques.

### 2.3. Motivation

Existing API call obfuscation and deobfuscation techniques are summarized above. As shown in Figure 2, API deobfuscation monitors all stages of existing obfuscation techniques, rendering them ineffective. IAT redirection and call site tampering introduce control flow jump to increase the execution distance between the call site and the system-loaded API, but there are still instructions to call the API in the “induction area” of the method (in Figure 1a,c). The API can be associated with the jump address obtained through call site monitoring (I and II in Figure 2). Position obfuscation, as shown in Figure 1b, increases the execution distance within the API by moving the code. However, current methods only move the first level of functions inside the API. This means that internal calls still provide enough information for deobfuscation to detect the API. As a result, deobfuscation can be successfully implemented by analyzing the functions called inside the API through the methods of hook monitoring and taint analysis, as shown in (III) and (IV) in Figure 2. The obfuscation scheme of DLL stolen is not practical because of its high overhead on program execution and vulnerability to monitoring.

In summary, existing API call obfuscation methods are ineffective against deobfuscation attacks. The main objective of these methods is to hide the address information during API calls to make it difficult to correlate virtual memory addresses with API names. However, this goal has not been achieved effectively so far.

The API address space is the basis for identifying API calls, so the goal of APIASO is to obfuscate the API address space information so that the analyst cannot establish a relationship between the virtual memory address of the function from the API call process and the API address space, and therefore cannot identify it as an API call.

Table 3 shows the differences between APIASO and other obfuscation techniques in terms of the means of obfuscation and the resistance to deobfuscation monitoring. APIASO obfuscates the entire process of the API call, while being able to withstand all types of existing deobfuscation techniques.
### Table 3. Differences between APIASO and other obfuscation techniques.

<table>
<thead>
<tr>
<th>Obfuscation Type</th>
<th>Resolution Process</th>
<th>Calling Process</th>
<th>Anti-Call Site Monitoring</th>
<th>Anti-Position Monitoring</th>
<th>Anti-Hybrid Monitoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAT redirection</td>
<td>×</td>
<td>√</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Position obfuscation</td>
<td>×</td>
<td>√</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Call site tampering</td>
<td>×</td>
<td>√</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>APIASO</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

### 3. API Address Space Obscurity Model

In this section, we introduce an obfuscation model for API address space obscurity, which focuses on two key processes in the API call process.

1. **API call space obfuscation**: This process involves the movement of API internal functions to user space for execution. The movement of the function requires a deeper analysis of the API’s internal functions. A function selection strategy is constructed by considering the call relationship, the function’s properties, the cost of the move, and the analyst’s experience.

2. **API name clue obfuscation**: This process involves building hash function generators and using more secure API address resolution methods and function movement schemes to obscure API name clues.

#### 3.1. Overview of the APIASO

Figure 3 illustrates the APIASO based on the guidance of the address space obscurity idea. When the program calls the API, it first executes the part of the API moved to the user code space, and then jumps to the system DLL space to execute the unmoved function inside the API. The function selection strategy involves making the function executed in the DLL space so it cannot be directly associated with the valid API. The collision-free hash function generator provides a separate hash scheme for each API name. The generated hash value is used as an encryption key to ensure the security of the function move, which makes the mapping relationship before and after the function move more complicated.

![Diagram of API address space obscurity.](image-url)

The obfuscation model discussed in this section protects various stages of API calls, and different technical points are discussed in subsequent sections. Specifically, the API
function selection strategy is described in Section 3.2, while Section 3.3 discusses the API address extraction method and function movement scheme. Table 4 shows the symbolic description of API address space obfuscation.

Table 4. Explanation of notations.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DASpace</td>
<td>DLL address space</td>
</tr>
<tr>
<td>APIEntry</td>
<td>API entry functions</td>
</tr>
<tr>
<td>APIInternal</td>
<td>API internal functions</td>
</tr>
<tr>
<td>G</td>
<td>Function call graph</td>
</tr>
<tr>
<td>AASpaceAPIij</td>
<td>API address space</td>
</tr>
<tr>
<td>A</td>
<td>Adjacency matrix</td>
</tr>
<tr>
<td>P</td>
<td>Reachable matrix</td>
</tr>
<tr>
<td>Level(fq)</td>
<td>Level of functions in the DLL address space</td>
</tr>
</tbody>
</table>

3.2. API Call Space Obfuscation

The goal of API call space obfuscation is to obscure the boundary between API and user functions so that the analyst fails to obtain information associated with known APIs. APIs typically have complex internal call relationships. As a result, obfuscation of key functions can effectively protect API calls, while obfuscation of non-key functions can result in higher overhead. The following chapter provides an analysis of the API’s internal call relationships and discusses the API call space obfuscation strategy.

Definition 1. Define the DLL address space as DASpace = {APIEntry, APIInternal, G DLL}, which comprises three parts: APIEntry = {f1, f2, · · · , fn}, includes the set of API entry functions called directly by the user; APIInternal = {fm+1, fm+2, · · · , fn}, includes the set of API internal functions called indirectly by the user; and m, which refers to the number of functions inside the DLL space that can be called directly by the user. n denotes the total number of functions inside the DLL space. G DLL represents the function call graph within the DLL address space, and is denoted as G DLL = {F DLL, E DLL}. F DLL in G DLL refers to the set of functions inside the DLL address space, i.e., F DLL = APIEntry + APIInternal.

Definition 2. Define each API address space as AASpaceAPIij = {(f1), (f1, · · · , fi, · · · , fi)}, G APIij, (1 ≤ i ≤ m), fi ∈ APIEntry indicates the functions that can be called directly by the user inside the API address space, {f1, f2, · · · , fm} ⊂ APIInternal indicates the set of internal call functions, and there is a direct or indirect call relationship between fi and f j. GAPIij denotes the APIi internal function call graph, denoted as GAPIij = {EAPIij}, (1 ≤ i ≤ m), where EAPIij denotes the set of functions, i.e., EAPIij = {(f1), (f1, · · · , fi, · · · , fi)}.

According to each API function call graph, GAPIij, (1 ≤ i ≤ m) can calculate G DLL, the calculation process is as follows:

\[
G_{\text{DLL}} = G_{\text{API}_1} \cup G_{\text{API}_2} \cup \cdots \cup G_{\text{API}_n}
\]  

(1)

The adjacency matrix can be used to describe the relationship between function calls. Based on the function call graph G DLL inside the DLL address space generates the adjacency matrix as A. If there exists a function fp calling a function fq (1 ≤ p ≠ q ≤ n) inside the DLL address space, then A pq = 1, otherwise A pq = 0. Obviously, the matrix A is a square matrix and the number of matrix ranks is equal to the number of DLL space functions n. The reachable matrix P and the adjacency matrix A are both Boolean matrices. \(P_{pq} = 1\) means that there is a call path from function fp to fq. The reachable matrix P can be obtained from the adjacency matrix A, which is calculated as follows:

\[
P = A^1 \vee A^2 \vee A^3 \vee \ldots \vee A^n, n = r(A) = c(A)
\]  

(2)
Definition 3. Define the level of functions in the DLL address space. For a function \( f_q \in AASpace_{API}(1 \leq i \leq m, m + 1 \leq q \leq n) \), if there exists another function \( f_p \in AASpace_{API}(1 \leq j, p \leq m, j \neq i) \) and there is a call path from \( f_p \) to \( f_q \), then the level of the function \( f_q \) is added by 1. Specifically, the API entry function called directly by the user is conventional of level 0, so for function \( f_q \in API_{internal} \in AASpace_{API} \) level is expressed as:

\[
\text{Level}(f_q) = \begin{cases} 
1 & \forall f_p \notin ASpace_{API} \text{ and } A_{pq} = 0 \\
> 1 & \exists f_p \notin ASpace_{API} \text{ and } A_{pq} = 1 \end{cases} \quad (m + 1 < q < n)
\]

Functions with a higher level are more difficult for analysts to associate with known APIs because they are called by more upper-level functions. Conversely, functions with a lower level are associated with fewer APIs, thereby enabling effective identification of the APIs called by the program. Therefore, the function level is an important criterion for moving the function to the user space.

When a function in the API address space \( AASpace_{API} = \{f_q\}, \{f_{i_1}, \cdots, f_{i_k}, \cdots, f_{i_m}\}, G_{API} \) is moved to the user space, the API entry function is moved directly to the user space because it is not called inside the address space. When a function \( f_{i_k} \) called from within the API is moved, moving it directly requires modifying the address of the \( f_{i_k} \) call instruction in the API address space. Modification of the instruction in the API address space will affect the correctness of other functions using the function. Therefore, it is necessary to move all the functions that dominate the function in the address space together to ensure the integrity of the call relationship.

The function \( f_{i_p} \) dominates the function \( f_{i_q} \) \((i_1 \leq i_p, i_q \leq i_k, p \neq q)\), which means that the API entry function \( f_{i_p} \) to \( f_{i_q} \) must pass through \( f_{i_p} \), which is denoted as \( f_{i_p} dom f_{i_q} \). If there exists \( i_p \) such that \( P_{i_q} = 1 \) and for any \( i_k \) \((i_1 \leq i_p, i_q, i_k \leq i_k, p \neq q \neq l)\), if it satisfies \( P_{i_q} = 1 \) and \( P_{i_k} = 0 \), then it is denoted \( f_{i_p} dom f_{i_k} \).

A necessary condition for API internal function movement: For the API address space \( AASpace_{API} = \{f_q\}, \{f_{i_1}, \cdots, f_{i_k}, \cdots, f_{i_m}\}, G_{API} \), if the function is the entry function \( f_{i_q} \), it will be moved directly; if the function is internal \( f_{i_k} \) \((1 < q < k)\), then all functions that satisfy \( f_{i_k} dom f_{i_q} \) are moved to user space together.

Based on the above definition, for a given API address space \( AASpace_{API} = \{f_q\}, \{f_{i_1}, \cdots, f_{i_k}, \cdots, f_{i_m}\}, G_{API} \), the function nodes that do not satisfy the address space obfuscation condition are pruned from the call graph \( G_{API} \).

1. Pruning high-level function nodes: As can be seen from Definition 3, there are low-level and high-level functions in the API address space \( AASpace_{API} \). Low-level functions are strongly associated with the upper API, and functions of Level 0 and 1 in the low-level functions are directly associated with the API itself, so such functions must be moved completely, while low-level functions of Level greater than 1 can be moved selectively according to the need for protection strength. Higher-level functions are called by many different APIs in the upper layers, and their calls are not sufficient to provide directly usable information for reconstructing the APIs. Moving them causes large memory and runtime overhead, so the higher-level function nodes are pruned on the call graph.

2. Adding special function nodes: In addition to low-level functions in the API address space \( AASpace_{API} \), there are also functions with certain special call relationships. Although these functions are not low-level functions, they still provide key information for accessing the API. For example, the CreateFileA function call eventually translates into a call to CreateFileW, and address space obfuscation is required for API calls with dependencies on xxxA and xxxW. In addition, some APIs call functions with names beginning with Nt exported from Ntdll.dll, using the system call number to enter the kernel. This class of API names provides information that can be used for deobfuscation. These functions may not be lower-level functions in that address space, so it is significant to recover this part of the pruned special function node.
(3) Adding bogus function node: low-level functions have a strong association with the API, and the movement of low-level functions in the address space of the protected API can hide the association, while the introduction of low-level functions in the address space of other APIs can increase the association with other APIs, thus misleading the analysts. Therefore, the low-level functions in other address spaces are chosen to be moved to the user space together as bogus function nodes.

According to the above conditions, the function call graph $G_{API}$ in the obfuscated API address space $AASpace_{API}$ is first pruned of its internal low-level function nodes; then, the special function nodes are restored, and bogus low-level function nodes are introduced on the pruned call graph, and the dominant function nodes and corresponding call edges are added to obtain the final call graph. It is worth noting that the API address space obscurity strategy provided in this paper is a controlled scheme, for example, the criteria for identifying low-level functions can be set to any level such as Levels 1, 2, and 3. The higher the set criteria, the more functions are moved, and with them the greater the obfuscation strength, and therefore the greater the overhead, which will be evaluated in Section 5.1.1 for the set value of the level.

3.3. API Name Clue Obfuscation

API address space obscurity gets the API code at runtime, and the position of the API code in memory can be obtained by API name resolution. The current API call obfuscation uses a name resolution method where the API name is mapped to a fixed length hash by a hash function, and the program executes to resolve the required API address by hashing the hash operation, and then matching the hash value. The main problem with this method is that the hash function is fixed and does not guarantee a diverse range of hash functions for more secure API address resolution.

To address the above problem, APIASO uses the more secure API name resolution method. To address the problem of using a single hash function, a hash function generator is designed to generate many fast and collision-free hash functions, and different API names are encrypted using different hash functions to ensure that a variety of hash functions are available. The details of this solution are described below.

The hash function generator is used to generate many lightweight collision-free hash functions, providing a variety of options for the API name resolution. API names are represented as sets $W = \{w_1, w_2, \ldots, w_l\}$ and the total number of API names is $k$. The 64-bit hash generation space $S$ takes values in the range $[0, 2^{64} - 1]$. We use a fast perfect hash function generation algorithm based on random graphs [20] to generate a hash function that satisfies the mapping of the set $W$ to the hash value space $S$ with no collisions, and the generic expression of the generated hash function is as follows:

$$h(w) = g(f_1(w)) + g(f_2(w))$$

where $f_i$ and $f_j$ are the operator functions that map the API names to the interval $[0, N - 1]$, and $N$ is the smallest integer that satisfies the collision-free hash function generation. $G$ is the mapping function that maps the results of the operations of functions $f_i$ and $f_j$ to the hash value space $S$. Given the set of operator functions $F = \{f_1, f_2, \ldots, f_l\}$, the execution of the algorithm can find the mapping function $g$. The specific algorithm execution process is as follows:

1. An integer $N$ greater than $k$ is selected randomly, and two hash functions $f_i$ and $f_j$ from the set $F$ are selected randomly afterward.
2. For each element $w_q$ in the set $W$, find $f_i(w_q)$ and $f_j(w_q)$.
3. An undirected graph $G$ is created, of which the vertices are defined by $f_i(w_q)$ and $f_j(w_q)$. Then, each pair of vertices $f_i(w_q)$ and $f_j(w_q)$ are linked up to obtain graph edges, in which each edge corresponds to each element $w_q$ of the set $W$.
4. $G$ is checked to see if it is acyclic, and if not, returns to Step 1.
5. \( N \) values are randomly selected in the hash generation space \( S \) and randomly assigned to the \( N \) edges of the graph \( G \) as the value of each element \( w_q \).
6. A randomly selected vertex is assigned a value of 0, and then a depth-first search is performed to traverse the graph \( G \) vertices. Correspondingly, the value of two vertices that share the same edge is assigned according to the hash value of this edge, such that the sum of the values of these two adjacent vertices equals to the hash value of the edge.
7. The sequence of vertices of the graph \( G \) and their assigned hash values form a mapping function \( g \). Thereby \( f_i \) and \( g \) constitute a collision-free hash function.

The time complexity of this algorithm is \( O(N) \) and the space required to store the generated functions is \( O(N \log N) \) bits, which is optimal for generating perfect collision-free hash functions [21]. It is almost impossible to involve all the APIs in the program’s API call process, so the size of the algorithm input \( k \) can be further optimized by adjusting \( k \) to the number of all API names in the DLL export table on which the program calls the APIs. Since only the API name hashes in the specified DLL import table need to be satisfied without collisions, this process greatly reduces the search space of the algorithm. The implementation of the above algorithm assigns a unique hash function to each API name, increasing the resistance of the API name resolution process to reverse analysis.

The API address space obscurity algorithm is shown in Algorithm 1. The inputs to the algorithm are the program \( P \), the DLL address space \( DASpace \), and the obfuscation threshold \( \epsilon \), where \( \epsilon \) represents the set criteria for the moved low-level function.

![Algorithm 1: Address Space Obscurity Algorithm](image-url)
The API collection of the program is first extracted, and API name trail obfuscation is performed to generate a collection of hash functions (22–23). Then, the API collection is traversed, and API call space obfuscation is performed for each API (24–26).

First, the call space obfuscation process prunes low-level function nodes and restores special function nodes (2–8); second, it adds bogus low-level function nodes (9–13); then adds function nodes on the call path according to the API internal function move requisites (14–18); and finally returns the obfuscated API internal call graph.

After performing call space obfuscation, a unique hash function is assigned to each API, and the obfuscated call graph, hash function, and name hash triads (27–29) are kept in the program, thus completing the obfuscation process.

After obfuscation, when the program executes the API call, it parses the API according to the triads and uses the saved hash to encrypt the mobile function to the user space for a more secure API call process.

4. System Implementation

The framework of LLVM [2] is an extensible program optimization platform that provides APIs for analyzing and modifying intermediate language code. In this section, we implement an obfuscation system on top of LLVM for automated API call obfuscation. Figure 4 demonstrates the system composition of the obfuscation system, which takes a C/C++ source program as input and outputs a binary file that has been obfuscated by the API calls. The obfuscation process is divided into the following four phases: (1) the front-end code compilation phase, (2) the obfuscated function addition phase, (3) the API call substitution phase, and (4) the code generation phase.

![Figure 4. Obfuscation system components.](image)

In the first stage, the Clang pre-section compiler converts the C/C++ source program into an intermediate representation of LLVM.

In the second stage, the obfuscated function is added to the intermediate language file of the program, and the obfuscated function completes the run-time API name and address space obfuscation.

In the third phase, the intermediate language is analyzed and the API calls that need to be protected are replaced.

In the fourth stage, the obfuscated “.bc” file is linked by the compiler to produce the final executable.

5. Experimental Evaluation

In this section, we perform an experimental verification of APIASO on a Windows 10 system with an Intel Core i7-9700 CPU @ 3.00 GHz and 32 G RAM. The APIASO automatically obfuscates API calls for programs. We evaluate it from the following two aspects:

(1) Model protection strength evaluation: We compare the advantages of APIASO with other API call obfuscation techniques in resisting API deobfuscation techniques, and compare the dynamic analysis resistance of programs before and after APIASO protection using online antivirus and sandbox platforms.
Model protection efficiency evaluation: We test large-scale code to evaluate the availability and accuracy of APIASO and the program time execution overhead, before and after obfuscation.

5.1. Model Protection Strength Evaluation

5.1.1. The Obscurity Degree of API Address Space

The obscurity degree of API address space is a key indicator of the effectiveness of API call obfuscation. To describe the ability of the obfuscation model to obfuscate the address space, we propose the concept of the obscurity degree of the API address space to quantitatively describe the degree of the obscurity of the API address space before and after obfuscation. Assuming that the program to be obfuscated is \( P \), the obfuscated program is represented as \( O(P) \). The obscurity degree of \( P \) is expressed as follows:

\[
APIASODegree(P) = \sum_{i=1}^{4} (-1)^{i-1} \frac{X_i - \min(X_i)}{\max(X_i) - \min(X_i)}
\]  

(5)

\( X_1-X_4 \) correspond to \( \text{LevelPer}(P) \), \( \text{Cost}(P) \), \( \text{ComplexIncreaseR}(P) \), and \( R(P) \), respectively.

\( \text{LevelPer}(P) \) indicates the proportion of high-level functions in DLL space executed when the program calls the API. The higher the percentage of high-level functions, the more low-level functions are moved to the user code space and the higher the degree of address space obscurity. \( \text{HighLevel}(P) \) indicates the accessed high-level functions. \( \text{Level}(P) \) indicates the total number of levels of the program. The \( \text{LevelPer}(P) \) calculation process is expressed as follows:

\[
\text{LevelPer}(P) = \frac{\text{HighLevel}(P)}{\text{Level}(P)}
\]

(6)

\( \text{Cost}(P) \) denotes the cost of obfuscation of APIASO, which is expressed as the ratio of the number of functions moved to the total number of functions. \( \text{MoveFun}(P) \) denotes the number of functions moved. \( \text{Fun}(P) \) denotes the total number of APIs and their internal functions in the program. The \( \text{Cost}(P) \) calculation procedure is expressed as:

\[
\text{Cost}(P) = \frac{\text{MoveFun}(P)}{\text{Fun}(P)}
\]

(7)

\( \text{ComplexIncreaseR}(P) \) represents the rate of increase in the complexity of the API call process after obfuscation. The complexity of the API call process is measured by the complexity of the API call relationship graph. \( \text{APICallComplex}(P) \) represents the complexity of the program API call process, then the \( \text{ComplexIncreaseR}(P) \) calculation procedure is expressed as:

\[
\text{ComplexIncreaseR}(P) = \frac{\text{APICallComplex}(O(P))}{\text{APICallComplex}(P)}
\]

(8)

The program remains semantically equivalent before and after API call obfuscation, but it becomes more difficult for the analyst to understand it. Ideally, any means of API call obfuscation can be cracked given enough time. In practice, however, program reverse analysis does not always result in the same level of difficulty as understanding the original program. \( R(P) \) is expressed as the experience of the reverse analyst. The higher the experience of the analyst, the closer the result of the reverse analysis is to the original program, and the easier it is to understand the program. The similarity between the reverse analysis result and the original program in the API call space in terms of data flow and control flow is used as the evaluation basis. The higher the similarity, the closer the reverse analysis result is to the original program, and the higher the reverse analyst’s experience
$R(P)$. $P^{-1}$ denotes the result of reverse analysis; $AC(P)$ and $AD(P)$ denote the control flow and data flow of API call space, respectively; and $R(P)$ denotes as follows:

$$R(P) = \text{Similarity}[AC(P), AC(P^{-1})] + \text{Similarity}[AD(P), AD(P^{-1})].$$  \hspace{1cm} (9)

As described in Section 3.2, APIASO is a method to control the strength of obfuscation. The threshold set by the lower level function determines the number of functions to move when obfuscating. To verify the effect of obfuscation under different thresholds, the $APIASODegree(P)$ is chosen as the judging basis for the obfuscation of six magnitudes of programs under Windows.

In Figure 5, the magnitude I is represented as a program containing 20 APIs, and the magnitude I to VI programs are incremented by 10 APIs. Figure 5 represents the results of $APIASODegree(P)$ versus execution time overhead for function levels set to 1, 2, 3, and 4. The horizontal coordinate represents the obscurity degree of the API address space at different level setting criteria, and the vertical coordinate represents the ratio of program execution time before and after program obfuscation. The results show that the best address space obfuscation is achieved at a low runtime overhead with a threshold of 2. Therefore, all subsequent experiments will be conducted with a threshold of 2.

![Figure 5. Address space obscurity at different levels.](image)

According to the previous section, the API call obfuscation methods include IAT tampering, basic block-level position obfuscation, function-level position obfuscation, DLL-level position obfuscation, and API call site tampering. We applied the above obfuscation methods and APIASO to six magnitudes of programs under Windows. The DLL space functions called by the program are recorded and the $APIASODegree(P)$, which is shown in Figure 6. The horizontal coordinate represents the obsfuscated program, and the vertical coordinate represents the $APIASODegree(P)$ after obfuscation. The results show that the APIASO is significantly higher than other obfuscation methods.

To visually describe the distribution of functions in memory during API calls, we track the execution of all functions specifically associated with CreateFileA calls. We count these functions and record their offset address and virtual memory address.

As shown in Figure 7, the Windows standard call procedure is compared with the calling procedure under the obfuscation of the other four types of API calls, with the vertical coordinate indicating the address offset of the function and the horizontal coordinate indicating the virtual memory address of the function. During a standard API call, only one call instruction exists in the user code space, so there is only one function associated with the API call. However, during API calls with call site tamper and IAT redirection, functions related to the obfuscation also exist in the user space in addition to the function...
containing the call instruction. No changes occur within the API code space. When it comes to API calls under position obfuscation, the code related to the entry function of CreatefileA exists in the user code space, while the entry function of CreatefileA in the API code space is hidden. Lastly, for API calls under APIASO, low-level, bogus, and special functions exist in the user code space along with the other obfuscation-related functions mentioned above. Only high-level functions remain in the API code space, while other key functions are hidden within the user code space.

![Figure 6. Degree of address space obscurity.](image)

![Figure 7. Function execution trajectory during API calls.](image)
Therefore, APIASO significantly obscures the boundary between user code space and API code space compared to other obfuscation techniques, which significantly increases the difficulty of API call analysis.

5.1.2. Anti API Deobfuscation Techniques

Table 5 shows the different API call obfuscation techniques used in ten popular packers’ software. To compare the advantages of APIASO with other API call obfuscation techniques in terms of resistance to deobfuscation detection, an application with the volume IV (containing 120 APIs) in the previous section is obfuscated using 10 packers and APIASO.

Table 5. Packer software corresponds to API obfuscation technology.

<table>
<thead>
<tr>
<th>No.</th>
<th>Tools</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Yoda’s Crypter</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Yoda’s Protector</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>TELock</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>ZProtect</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Enigma</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Armadillo</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>Obsidium</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>PESpin</td>
<td>1, 2</td>
</tr>
<tr>
<td>9</td>
<td>PELock</td>
<td>1, 2</td>
</tr>
<tr>
<td>10</td>
<td>PEP</td>
<td>1, 3</td>
</tr>
</tbody>
</table>

Section 2.2 provides a summary of the various deobfuscation techniques currently used with APIs in Table 2. For our experimental evaluation, we selected four representative techniques: Scylla, PinDemonium, QuietRIATT, and RePEconstruct. Scylla and PinDemonium employ call site monitoring, while QuietRIATT utilizes a position monitoring approach and RePEconstruct uses a hybrid monitoring approach.

In Figure 8, a radar chart is presented to show the deobfuscation capabilities of four different techniques when faced with various API call obfuscation methods. The results of our experiments were that none of these techniques were able to successfully deobfuscation the program when APIASO was used. Furthermore, the recovered API information contained errors, including bogus APIs that had been introduced, leading to an increase in the overhead associated with analysis for the analyst.

The experimental results are predictable from the principles of the four deobfuscation techniques. Scylla and PinDemonium collect the target addresses of calls and jump instructions from the program memory under APIASO obfuscation, but since these cannot be associated with the API addresses exported by the DLL, they cannot identify the obfuscated APIs. QuietRIATT sets hooks at API entry points that may be called by programs. Programs under APIASO obfuscation do not execute API entry functions, and therefore cannot be logged by QuietRIATT. RePEconstruct uses a binary instrumentation tool to record instructions that jump into DLL space, while the destination addresses of calls and jump instructions in memory are collected. The destination addresses of instructions that do not exist in the memory of programs under APIASO obfuscation are API addresses, and the addresses monitored in DLL space cannot be associated with valid APIs. Therefore, RePEconstruct cannot implement deobfuscation.
Figure 8. API deobfuscation Capability Radar Chart.

5.1.3. Sandbox and Antivirus Platform Detection

Analysts often use the results of online sandbox detection as basics for further reverse analysis, while API calls are an essential metric in detection. In this section, programs protected by APIASO are analyzed using VirusTotal (https://www.virustotal.com/gui/home/upload, accessed on 13 August 2022), Cuckoo (https://cuckoosandbox.org/, accessed on 13 August 2022), and Sandboxie (https://sandboxie-plus.com/, accessed on 13 August 2022) to compare the differences in output results between platforms before and after obfuscation. We collect 100 publicly available malicious programs with source code from GitHub, then submit the obfuscated malicious programs to VirusTotal and Cuckoo to compare the difference in output results between platforms before and after obfuscation.

VirusTotal uses a total of 78 different antivirus detection tools to mark submitted malicious programs as malicious or benign based on each security vendor’s judgment; the more security vendors mark them as malicious, the less resistant the program is to analysis. The analysis results are shown in Figure 9. VirusTotal detects significantly fewer security vendors for obfuscated malicious programs due to APIASO obfuscating the API call information of malicious programs, with nearly half of the security vendors marking them as benign compared to before obfuscation.

Cuckoo is a dynamic malicious program analysis sandbox that calculates a malicious rating for each submitted program based on the hit signature: less than 1.0 is benign, 1.0–2.0 is a warning, 2.0–5.0 is malicious, and above 5.0 is dangerous. Figure 10 Indicates the results of Cuckoo’s detection. APIASO successfully reduced Cuckoo’s score, with the obfuscation-protected programs all having malicious scores below 5.0 and significantly lower malicious behavior levels, all dropping below the malicious level.
We build Sandboxie in an experimental environment to run six API-weight programs after APIASO obfuscation. Sandboxie is a sandbox-based isolation program for Windows NT-based 32-bit and 64-bit operating systems. It was developed by David Xanatos since it was open sourced, and before that, it was developed by Sophos. It creates a sandbox-like isolated operating environment in which applications can be run or installed without permanent modification of local or mapped drives. Unobfuscatedly, the four programs are run in Sandboxie’s isolated sandbox, and upon execution, an event log of that program’s execution is generated, recording events such as file and registry operations. As a comparison, the API calls obfuscation techniques are applied to protect the above programs. The principle of the Sandboxie detecting API is to hook in the API area loaded by the system; when the program is executed, the API called will be recorded, so the method of position obfuscation cannot achieve the effect of API monitoring by hook monitoring. The results in Table 6 show that the program behavior under the protection of APIASO and position obfuscation cannot be captured, while several other obfuscation methods can be effectively observed in the event log.

![Graph](image.png)

**Figure 9.** VirusTotal analysis results.

![Graph](image.png)

**Figure 10.** Cuckoo analysis results.
Table 6. Sandboxie behavior monitoring results.

<table>
<thead>
<tr>
<th>Type</th>
<th>Program</th>
<th>I</th>
<th>II</th>
<th>III</th>
<th>IV</th>
<th>V</th>
<th>VI</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAT redirection</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Position obfuscation</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Call site tampering</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>APIASO</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
</tbody>
</table>

5.2. Model Protection Effect Evaluation

The API address space is obscured using API code extracted from the system DLL, while the accuracy of this process still needs to be verified. The Windows API includes thousands of functions that can be called, and Microsoft officially classifies these functions into the following broad categories: basic services, component services, user interface services, graphics multimedia services, messaging and collaboration, networking, and web services. To cover the above types of services, the following 12 types of programs are implemented by using over 1000 Windows APIs: file handlers, network programs, message handlers, printers, text and font functions, menu handlers, bitmap, and raster arithmetic programs, drawing programs, device scenario programs, hardware and system programs, process and thread programs, and control and messaging programs.

Figure 11 describes this verification process. The test process is based on LLVM and is divided into the following stages: (I) converting the source program containing the API calls into the corresponding intermediate language file Src.bc; (II) extracting the API code from the system DLL file using a decompiling tool; (III) completing the API replacement at the intermediate language level with LLVM, link the intermediate file; (IV) outputting the unobfuscated binary; (V) running the binary before and after the obfuscation to compare the two functions and verify the correctness of the API execution. In the end, the experimental results confirm the availability of APIASO.

![Figure 11. APIASO availability testing.](image)

The APIASO is compared with the above three API call obfuscation methods to evaluate the program execution efficiency, and the obfuscated program is selected for six API magnitudes.

For each test program, to accurately measure the running time of the program, each program was looped 100 times, and the average running time before and after the obfuscation was calculated.
Figure 12 shows the time overhead before and after obfuscation for APIASO versus the other three API call obfuscation methods. The x-axis represents each test program, and the y-axis represents the ratio of post-obfuscation to pre-obfuscation time overhead. The results show that APIASO is close to the other obfuscation methods in terms of time overhead, and the overall obfuscation time overhead is no more than 20%.

6. Discussion

A perfect solution for obfuscating API calls is far-fetched, and there is a constant cat-and-mouse game between program protectors and attackers. In this section, we discuss the following possible attack methods against APIASO and the corresponding countermeasures:

Kernel-level hook: APIs are divided into user-level and kernel-level parts, and an attacker may use a kernel-level API hook for API monitoring purposes, but a kernel-level hook is not sufficient for API monitoring. This is because there is no bijective mapping [22] between the user-level API and the kernel-level API. On the one hand, some user-level APIs such as path-related APIs and DLL management APIs (e.g., GetProcAddress) provide user-level services exclusively, which means that they do not call any kernel-level APIs at all; on the other hand, the kernel part of the API (e.g., NtCreateFile) serves multiple APIs at the upper level, and it is difficult to fully recover upper-level API calls through kernel-level API Hooks alone. The hook is difficult to fully recover upper-level API call information.

Instruction sequence similarity matching: APIASO copies the API to the user code space for execution, still retaining the original control flow structure of the API. With the help of API identification techniques (BinShape [23], IDA FLIRT [24], etc.), the runtime memory is compared with known APIs for similarity, and the called APIs can be found. For the above problem, control flow obfuscation techniques at the binary level can be introduced to perform control flow transformations and instruction transformations on the internal functions of the moved APIs to achieve resistance to similarity analysis purposes.

Monitoring NX bits: Seokwoo [25] et al. proposed monitoring the NX bit to detect the copying process of API code and monitor the DLL page memory access rights during program runtime. Since all API position obfuscation methods must move the API code through read and write operations, the API can be associated with the copied API memory address through read and write operations. However, read and write operations are frequent during program runtime, and restoring the API through read and write operations alone is difficult. Alternatively, the DLL can be read as a file during runtime, making the acquisition of API code independent of the DLL in the system load space.

7. Conclusions

In this paper, we systematically analyze existing API call obfuscation and deobfuscation techniques. It is shown that none of the existing API call obfuscation models can effectively resist the attacks of API deobfuscation techniques. The reason for the poor
obfuscation effect of the existing API call obfuscation models is the insufficient obscure of the API address space. Therefore, we propose and construct the API address space obfuscation model. Compared with existing API call obfuscation schemes, APIASO obfuscates the API resolution and calling process with higher security. The experiments show that, after obfuscation, the API address space obscurity increases by more than two times, the detection rate of VirusTotal, etc. decreases by more than four times, the detection rate of deobfuscation techniques such as Scylla, etc. is zero, and the increase in obfuscation overhead is not more than 20%. The above results show that APIASO has better obfuscation effect and practicability.

In future work, the generalization of the obfuscation system still needs to be refined. The prototype code obfuscation system implemented in this paper was carried out when the program was available in source code, and future work will be carried out by applying the obfuscation system to the binary level. The conversion technology from binary code to LLVM intermediate language is already available, which provides technical support for code obfuscation at the binary level. In addition, the obfuscated prototype system currently only accepts C/C++ source code as input, but the LLVM platform supports a wide range of high-level programming languages, which can be converted into a unified intermediate language form. Therefore, future work will need to test code in many different programming languages to improve the generalizability of the obfuscation system.

**Author Contributions:** Conceptualization, Y.L. and F.K.; Methodology, Y.L. and H.S.; Investigation, Y.L. and X.X.; Writing—original draft preparation, Y.L.; writing—review and editing, Y.L., Y.Z. and R.S.; Supervision, F.K.; Funding acquisition, H.S. All authors have read and agreed to the published version of the manuscript.

**Funding:** This paper is supported by the National Key R&D Program of China, grant number 2019QY1305. The authors would like to acknowledge them.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Informed consent was obtained from all subjects involved in the study.

**Data Availability Statement:** The data of this study are included within the article.

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

**References**

6. Roundy, K.A.; Miller, B.P. Binary-code obfuscations in prevalent packer tools. ACM Comput. Surv. (CSUR) 2013, 46, 1–32. [CrossRef]


Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.