Obfuscation-Resilient Executable Payload Extraction From Packed Malware

Binlin Cheng, Ming Jiang, Erika A Leal, Haotian Zhang, Jianming Fu, Guojun Peng, Jean-Yves Marion

To cite this version:

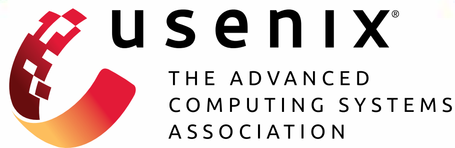
Binlin Cheng, Ming Jiang, Erika A Leal, Haotian Zhang, Jianming Fu, et al.. Obfuscation-Resilient Executable Payload Extraction From Packed Malware. 30th Usenix Security Sympoisum, Aug 2021, Virtual, United States. hal-03549482

HAL Id: hal-03549482
https://hal.science/hal-03549482

Submitted on 31 Jan 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Obfuscation-Resilient Executable Payload Extraction From Packed Malware

Binlin Cheng, *Hubei Normal University & Wuhan University;* Jiang Ming, Erika A Leal, and Haotian Zhang, *The University of Texas at Arlington;* Jianming Fu and Guojun Peng, *Wuhan University;* Jean-Yves Marion, *Université de Lorraine, CNRS, LORIA*

https://www.usenix.org/conference/usenixsecurity21/presentation/cheng-binlin

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.

Obfuscation-Resilient Executable Payload Extraction From Packed Malware

Binlin Cheng[*†]
Hubei Normal University & Wuhan University, China
binlincheng@163.com

Jiang Ming[*‡]
The University of Texas at Arlington, USA
jiang.ming@uta.edu

Erika A Leal and Haotian Zhang
The University of Texas at Arlington, USA
{erika.leal,haotian.zhang}@mavs.uta.edu

Jianming Fu[†] and Guojun Peng[†]
Wuhan University, China
{jmfu,guojpeng}@whu.edu.cn

Jean-Yves Marion
Université de Lorraine, CNRS, LORIA, F-54000 Nancy, France
jean-yves.marion@loria.fr

Abstract

Over the past two decades, packed malware is always a veritable challenge to security analysts. Not only is determining the end of the unpacking increasingly difficult, but also advanced packers embed a variety of anti-analysis tricks to impede reverse engineering. As malware's APIs provide rich information about malicious behavior, one common anti-analysis strategy is API obfuscation, which removes the metadata of imported APIs from malware's PE header and complicates API name resolution from API callsites. In this way, even when security analysts obtain the unpacked code, a disassembler still fails to recognize imported API names, and the unpacked code cannot be successfully executed. Recently, generic binary unpacking has made breakthrough progress with noticeable performance improvement. However, reconstructing unpacked code's import tables, which is vital for further malware static/dynamic analyses, has largely been overlooked. Existing approaches are far from mature: they either can be easily evaded by various API obfuscation schemes (e.g., stolen code), or suffer from incomplete API coverage.

In this paper, we aim to achieve the ultimate goal of Windows malware unpacking: recovering an executable malware program from the packed and obfuscated binary code. Based on the process memory when the original entry point (OEP) is reached, we develop a hardware-assisted tool, *API-Xray*, to reconstruct import tables. Import table reconstruction is challenging enough in its own right. Our core technique, *API Micro Execution*, explores all possible API callsites and executes them without knowing API argument values. At the same time, we take advantage of hardware tracing via Intel Branch Trace Store and NX bit to resolve API names and finally rebuild import tables. Compared with the previous work, API-Xray has a better resistance against various API obfuscation schemes and more coverage on resolved Windows API names. Since July 2019, we have tested API-Xray in practice to assist security professionals in malware analysis: we have successfully rebuilt 155,811 executable malware programs and substantially improved the detection rate for 7,514 unknown or new malware variants.

1 Introduction

Over the past two decades, malware is always one of the top cyber threats that can cause catastrophic damage. In 2020, over 350,000 new malicious programs worldwide are identified every day [39]. McAfee Lab estimates that malware revenue can reach multiple billions of dollars by 2020 [45]. Driven by huge financial gains, malware authors typically obfuscate their programs to circumvent malware detection. Among different obfuscation technologies, binary packing is the most prevalent one adopted by Windows malware [49, 55, 74, 86], because it protects the original code from static analysis and has a right balance between strength and performance [1, 6]. The trend of binary packing development reveals two salient features. **First**, when a packed malware starts running, the attached unpacking routine will pass through multi-layers of self-modifying code until the original entry point (OEP) of the payload is reached [11, 73]. **Second**, complicated packers also incorporate multiple anti-analysis methods (e.g., anti-debugging, anti-hooking, and anti-sandbox) to hinder both automated and manual unpacking attempts [63].

On the defender side, generic binary unpacking has generated a large body of literature [8, 15, 35, 38, 51, 52, 60]. Most of them rely on memory access tracing to find the reach of OEP and then dump the current process memory as the unpacked program. The recent progress in this direction, BinUnpack [15], proposes an effective heuristics to quickly determine the end of multi-layer unpacking without heavy memory access tracing. Despite this, in BinUnpack's large-scale evaluation, the authors have admitted that many unpacked

[*]Both authors contributed equally to the paper.

[†](1) Key Laboratory of Aerospace Information Security and Trust Computing, Ministry of Education; (2) School of Cyber Science and Engineering, Wuhan University.

[‡]Corresponding authors: jiang.ming@uta.edu.

malware samples cannot run, which weakens the utilization of unpacked code in further malware analysis. This lack of executability is not just BinUnpack that faces issues, and it is actually fairly common for existing generic unpacking solutions [8, 35, 38, 51, 52, 60]. The root cause is the unpacked code's import tables, which store the metadata of imported APIs, are corrupt.

To perform malicious behavior (e.g., code remote injection and C&C communication) in diverse Windows systems, malware samples interact with Windows OS through user-level Windows APIs [27, 29, 53]. The import table structures in a PE* file's header store the information about Windows APIs that the PE file requires to execute. In particular, Import Address Table (IAT) is an address lookup table for calling Windows APIs; API callsites in a PE file refer to the IAT via indirect calls/jumps.† Note that the IAT does not take effect until the program is loaded into memory. Another table, Import Name Table (INT), containing API names corresponding to IAT entries, can be treated as the IAT's index. Using the INT, Windows PE loader fills each IAT entry with the associated Windows API's virtual address. Since these two tables are referenced from the PE header, examining these data from a malware sample's header yields information about this malware's capability. For example, a malware instance that imports functions from advapi32.dll is likely to access or change the registry; "CreateRemoteThread" API is often misused for the purpose of process injection [37]. The list of loaded APIs is necessary and of great practical significance for understanding malware behavior [3, 4, 32, 65]. Especially for a new malware variant, its instructions may not match any known malware signatures, but API calls can still provide valuable insight into its malicious intention [70].

Unfortunately, binary packer developers are also aware of the value of imported APIs. They reduce the wiggle room for security analysts by not using the standard API resolution in the packed PE file. Two key factors amplify the attackers' advantage. First, they delete INT & IAT entries and dereference both of them from the PE header, so that these two import tables become unavailable for analysis. Second, to sustain the original functionality after unpacking, the attached unpacking routine works in an ad-hoc way to customize a new IAT before the execution of the original code. As our study shows in §3, this custom IAT can contain misleading entries to hide the real names of invoked APIs. As a result, given the process memory produced by a generic unpacking tool, a disassembler fails to recognize API names at API callsites (e.g., call [f_1]), because IAT entries (e.g., [f_1]) do not *directly* point to the related APIs in a DLL. In addition, without the INT as an index, Windows PE loader cannot load correct API addresses of the target system into the IAT, and therefore the unpacked program has no executability in a Windows system.

*Portable Executable (PE) is an executable file format in Windows OS.
†e.g., call [f_1], and f_1 points to an entry of the IAT.

The goal of import table reconstruction is to resolve invoked API names and reconstruct the removed INT & IAT for the unpacked program. Therefore, a disassembler can recognize imported APIs to facilitate static analysis. Furthermore, the unpacked program becomes a "working PE" that can be successfully executed. Since most of the anti-analysis tricks embedded in a packer have been removed, this working PE unleashes the power of malware dynamic analysis. We name the process memory when the unpacking algorithm reaches the original entry point of the packed program as ***OEP memory***. The research question of recovering a working executable file is, *given OEP memory, how to reason about the real API name corresponding to each API callsite*. Especially, different API obfuscation schemes (e.g., stolen code and ROP redirection) may go through a lengthy call/jump chain before reaching the real API code, rendering manually resolving API names tedious and error-prone.

Researchers have realized the significance of import table reconstruction [2, 15, 17, 40–42, 44, 62, 69, 73, 81]. However, they do not cover all API obfuscation methods that we present in this work. The current solutions can only resolve partial API names either for statically identifiable targets or limited targets in a single execution trace. No work claims to resolve Windows API names for an unpacked program completely. Besides, many approaches track the target of an API call using API hooking [62] or dynamic binary instrumentation [17, 40–42, 73], but malware can fingerprint these monitoring environments.

In this paper, we aim to bridge the gap in the generic binary unpacking domain. Our technique, named *API-Xray*, is a hybrid static-dynamic analysis towards complete import table reconstruction. To transparently collect runtime control flow targets, we take a less intrusive approach with no code injection via hardware-assisted mechanisms: Intel Branch Trace Store (BTS) and NX bit [20]. More concretely, given an unpacked program's OEP memory, we first perform static analysis to explore all potential API callsites. Then, our proposed "API Micro Execution" enforces executing each potential API callsite without requiring concrete function arguments. At the same time, we track the target address of an API call with the help of BTS's branch tracing capability. In addition, we also enable NX bit to interrupt the execution when the complex control flow finally reaches the target API. After that, we further analyze BTS record and determine the real API addresses that we need. At last, we associate the obtained API addresses with the corresponding API names, rebuild INT & IAT entries, and restore the PE header. API-Xray complements the state-of-art binary unpacking tools and frees security analysts from the burden of manually piecing together the tedious steps of import table reconstruction.

We conduct a set of experiments to evaluate the efficacy of API-Xray. We first compare API-Xray with representative related work (e.g., S&P'15 [73] and CCS'18 [15]) on the prevalent packers that contain API obfuscation. API-Xray is

the only one that can defeat different API obfuscation schemes and reconstruct import tables entirely in all cases. Compared with the other two common hardware tracing mechanisms, namely Last Branch Record and Intel Processor Trace, we demonstrate that BTS is the only right option for import table reconstruction. At last, we report our experience after testing API-Xray in practice: 1) we have successfully rebuilt 174,285 malware's import tables, and 149,488 of them are labeled as "Malicious" by at least one malware sandbox. 2) for 7,514 unknown or new malware variants, the output of API-Xray can substantially increase the accuracy of malware detection. In summary, this paper makes the following contributions:

- The success of import table reconstruction hinges on the deep understanding of API obfuscation. Our in-depth study unveils new API obfuscation schemes that are not public before. Our work serves as a baseline for evaluating the effectiveness of future work (§3).
- Our proposed "Hardware-Assisted API Micro Execution" is the first approach towards recovering an executable program from the packed code. Our work exhibits strong resistance to API obfuscation and significantly lightens the burden of security professionals (§4).
- We have evaluated API-Xray extensively with a large-scale dataset, including prevalent and custom packers. API-Xray maintains a high success rate consistently in all cases (§5).

2 Background and Related work

Given a packed malware sample, our work assumes that security analysts have already found the original entry point (OEP) of malware and obtained the process memory at that time (i.e., OEP memory). Note that most of the current generic unpacking tools can meet this prerequisite. This paper tackles binary packer's API obfuscation, which deters security analysts from the further analysis of the unpacked code.

Various API obfuscation schemes are prevalent in packers. We first introduce the background information needed to understand this challenging problem. Then we discuss the limitations of existing import table reconstruction methods. At last, we introduce the work most germane to our hardware-assisted control flow monitoring. Regarding the complexity of advanced packers and the status quo of generic unpacking techniques, interested readers are referred to three papers: CSUR'13 [63], S&P'15 [73], and CCS'18 [15].

2.1 Binary Packers Avoid Using Standard API Name Resolution

The Role of Import Tables. Figure 1 illustrates an example of standard API resolution. The header of a PE file contains two import tables: import name table (INT) and import address table (IAT). These two tables contain names and addresses of APIs that need to be imported from a specific DLL (e.g., kernel32.dll), respectively. Since the compiler is unaware of imported API addresses at compilation time, IAT entries are first filled with placeholders temporarily (❶). Note that API names in the INT and API addresses in the IAT are maintained in the same order (❷). When the executable file is loaded, Windows PE loader is responsible for mapping the required DLLs into the memory address space of the application, and then it fills each IAT entry with the API address according to the item order of INT (❸). After that, the IAT begins to take effect as an address lookup table for calling Windows APIs. An API callsite in the PE file refers to the IAT via an indirect control flow instruction (e.g., call [f_1]): it reads the API address from an IAT entry and then jumps to the target (❹). This design ensures the compatibility with different Windows OS versions and address space layout randomization, in which *each imported API is very likely to have a different address at every execution* [64].

Delete Import Tables. The metadata of imported APIs is detailed enough to allow a security analyst to estimate whether a particular sample is malicious [70]. Therefore, when packing malware code, a binary packer erases both INT and IAT entries and makes these two tables unreachable from the malware's header. At the same time, the packer maintains a list of removed APIs in the attached unpacking routine. When the packed malware starts running, the unpacking routine, which runs an unpacking algorithm to recover the original code, has to rebuild a new IAT before the execution of the original code; otherwise, it will lead to an execution crash. The most convenient way is to use "LoadLibrary" and "GetProcAddress" APIs [63]: calling these two functions can explicitly load a DLL and obtain an API address at run time.

Hide Invoked API Names. In addition to the removal and deference of import tables, advanced packers also adopt other methods to obfuscate the use of API calls and the control flow ❹ in Figure 1. The purpose is to disconnect API callsites from corresponding target API names. Symantec security response [70] makes the first study of API obfuscation used in the wild. They find that some packers generate the target address of API via sophisticated instructions such as "push-calc-ret" and "Structured Exception Handler" to impede static analysis. Kawakoya et al. [42] and their follow-up work [41] formally define the concept of API obfuscation and introduce several specific patterns of API obfuscation, such as IAT redirection and stolen code. In §3, we will present our in-depth study to inspect the multiple types of API obfuscation that mislead the control flow ❹, including new API obfuscation schemes that have been swept under the carpet.

Anti-Static Analysis. API obfuscation combines the removal of import tables and the methods of hiding invoked API names, with the purpose of concealing a program's functionality. Even though security analysts obtain the OEP memory,

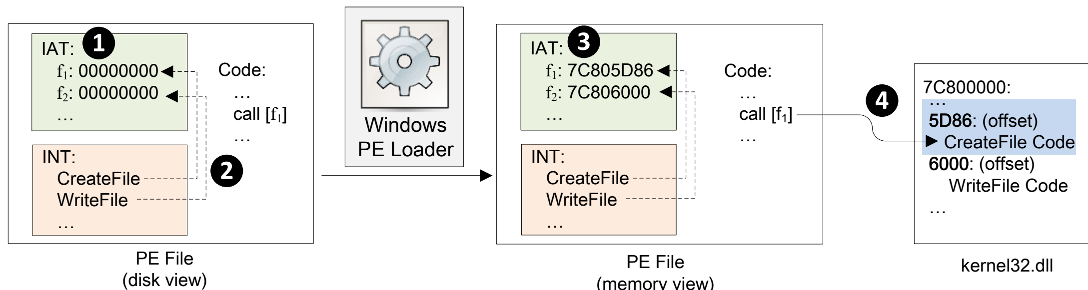


Figure 1: The example of standard API resolution. The "00000000" in the disk view represents a placeholder; Windows loader will replace it with an API address (e.g., "7C805D86") at program loading time according to the INT.

further malware analysis would still be difficult, if not impossible. As addresses stored in the IAT do not directly point to correct API entry points, a disassembler is unable to recognize API names. Therefore, when analyzing a new unpacked malware sample, the only available resource for security analysts is low-level assembly code, lacking semantic abstractions represented by API calls.

Anti-Dynamic Execution. Besides, API obfuscation impedes the reconstruction of a fully executable PE file—this is the ultimate goal of generic binary unpacking. Advanced packers have embedded various heuristics to detect the existence of malware dynamic analysis environments (e.g., debugging, hooking, and sandbox). The state-of-the-art unpacking tool, such as BinUnpack [15], is immune to these anti-dynamic-analysis methods and can quickly produce a memory dump as the unpacked program. However, if the removed INT cannot be recovered, Windows PE loader has no idea about which API addresses should be filled in the IAT, and thus security analysts cannot independently run the unpacked malware to observe its intended malicious behavior.

2.2 Limitations of Existing Work

To facilitate malware analysis, a complete binary unpacking solution has to restore the original code as well as reconstruct import tables. Typically, an import table reconstruction starts after the unpacking tool captures the OEP memory, and it contains three major steps [41]: 1) identify possible API callsites from the process memory; 2) resolve API names according to API callsites, and 3) restore both INT and IAT in the PE header (just like ❶ & ❷ in Figure 1). Among them, the most challenging step is **API name resolution**, as IAT entries may not directly point to target APIs, and all of the existing approaches fail to resolve API names completely. Table 1 summarizes different import table reconstruction approaches in terms of memory static analysis, dynamic analysis, and hybrid analysis.

Memory Static Analysis. The approaches in this category perform static scanning on OEP memory to identify indirect call (e.g., call [f_1]) and indirect jump (e.g., jmp [f_2]) instructions. Each target address of these instructions is considered as an entry of the IAT. Then, they relate all addresses in the IAT with corresponding API names in the INT. In particular, they match IAT entries with the address calculated from a DLL's base address and the offset of each export API in the DLL's export address table. In this way, they resolve the API name for each IAT entry. The latest generic unpacking work in CCS'18 [15] also adopts a similar style to reconstruct import tables. However, all of these static methods suffer from the same limitation: they can only recognize statically identifiable targets; the different API obfuscation methods that we will present in §3 can easily nullify them.

Table 1: Different import table reconstruction approaches. §5.1 compares API-Xray with the three approaches in bold.

Class	Approach
Memory Static Analysis	**BinUnpack** [15], Scylla [2], Eureka [69], RePEF [81], PinDemonium [50], Arancino [60]
Dynamic Analysis	**Ugarte-Pedrero et al**. [73], API Chaser [42], API Deobfuscator [17], QuietRIATT [62], tf_impscan [41], Secure Unpacker [40]
Hybrid Analysis	**RePEconstruct** [44], API-Xray

Dynamic Analysis. Another direction reconstructs import tables at run time. S&P'15 paper [73] achieves this by instrumenting indirect calls/jumps and grouping the memory addresses used in these instructions. Both Secure Unpacker [40] and QuietRIATT [62] use hooking-based methods to identify target API. They assume no matter what API obfuscation techniques the packer used, the control flow will be transferred to the API code eventually. Therefore, they set hooks at the entries of APIs that the packed sample is very likely to call. Once the control flow arrives at the hooked API's entry point, they can determine the target API's name. Unfortunately, stolen code, which we will further discuss in §3, defies the assumption embodied by hooking-based methods. To overcome this limitation, both Kawakoya et al. [41, 42] and Seokwoo et al. [17] use taint analysis to trace the code copy operation, which is necessary to complete the stolen code obfuscation. As the stolen code shares the same taint

Table 2: The comparison of hardware control flow tracing mechanisms. "Yes" in the "Completeness" column means it can monitor all kinds of control flow deviation instructions, including jmp, cjmp, call, ret, and exception.

Mechanism	Completeness	Size Limit	Overhead	Online/Offline
LBR	Yes	Yes	Low	Online
BTS	Yes	No	High	Online
IPT	No	No	High	Offline

tag with the source API code, when the program executes the stolen code, the attached taint tag can decide which API code is actually executed. The disadvantage of dynamic-based methods is also apparent—they can only resolve API names in a single execution path each time. Single-path API coverage cannot guarantee the executability of malware in a new Windows OS, because non-identical environments are likely to trigger a different execution path. Furthermore, as most of the dynamic analysis environments are not transparent, malware can counter them via anti-sandbox and anti-debug heuristics.

Hybrid Analysis. RePEconstruct [44] takes a weak hybrid analysis style to resolve API names. Like S&P'15 paper [73], RePEconstruct leverages dynamic binary instrumentation (DBI) to record the branch instructions that jump to dynamically loaded modules. In addition, it also takes another round of memory static scanning to recognize the APIs that are not executed at run time. However, its memory static scanning does not consider API obfuscation. By contrast, API-Xray weaves static and transparent dynamic analyses in a compatible manner that amplifies each other's benefit.

2.3 Control Flow Monitoring via Hardware

Multiple software security tasks require control flow monitoring to block anomaly intrusions, such as defending ROP attacks [59, 89] and preventing kernel malware [46, 79]. The software-based monitoring typically relies on a DBI platform (e.g., Pin [48] or DynamoRIO [10]) to record control flow transfers. However, DBI tools do not keep the code under execution intact, and thus their instrumentation environments are easy to be detected [25]. In contrast, hardware-based monitoring overcomes the limitation of lacking transparency; it leverages modern CPU features to record control flow, requiring no code injection. For modern Intel processors, the mechanisms to trace branch instructions include Last Branch Record (LBR), Branch Trace Store (BTS), and Intel Processor Trace (IPT). We will further evaluate these three mechanisms in §5.1, but for now, we would like to remind readers that BTS is the only option for import table reconstruction. Table 2 shows the different features of these three hardware tracing mechanisms.

LBR. LBR can record 16 or 32 most recent branch pairs (source and target) into a register.[‡] LBR is very fast since it directly accesses CPU registers, but LBR is also limited by the maximum number of branches that it can record at one time [82]. kBouncer [59] is the first work to use LBR to prevent ROP attack. At each system API invocation, kBouncer checks the proposed control-flow integrity (CFI) policy against LBR stack. Later, ROPecker [16], CFIGuard [88], and PathArmor [75] extend kBouncer [59]'s idea to prevent ROP attacks with the help of LBR. However, due to the limited size of LBR stack (16 or 32), an attacker can still circumvent LBR's monitoring [12, 24, 34].

BTS. BTS is more flexible than LBR. BTS records all kinds of branch pairs (source and target) into a memory buffer, and users can determine the memory buffer's size and location. Unlike LBR that overwrites the data when LBR stack is full, BTS can be configured to halt the application when the recording buffer is full, or when a predefined exception is triggered. Then, the user saves BTS buffer's record, resets it, and then resumes BTS's monitoring. In this way, BTS is able to record complete control flow transfers, but at the cost of higher overhead than LBR. To prevent ROP attacks, CFIMon [83] and Eunomia [87] leverage BTS to detect illegitimate branch pairs. Recent work [9] proposes a general BTS-based control flow monitoring framework, which can be extended to perform different analysis tasks, such as control flow graph reconstruction and ROP detection.

IPT. Different from both LBR and BTS, IPT is initially designed for offline performance analysis and software debugging. IPT efficiently captures control flow traces online, but it is at the cost of the orders-of-magnitude slowdown in offline decoding. In addition, IPT does not trace all types of control flow transfer instructions—unconditional direct branches (e.g., direct jump and direct call) are not logged [20]. Both GRIFFIN [31] and FlowGuard [47] transparently enforce fine-grained CFI policy using IPT.

3 Deep Inspection of API Obfuscation

In this section, we conduct an in-depth study to demystify API obfuscation schemes that can hide the names of invoked APIs. We manually analyzed all of the 29 prevalent packers tested in BinUnpack's paper [15]. Upon further investigation, we find that 12 out of 29 packers obfuscate the control flow between API callsites and target API entry points (i.e., the control flow ❶ in Figure 2(a)). These 12 packers (e.g., Themida [57], Enigma [72], and Obsidium [54]) represent advanced packers that incorporate multiple anti-analysis methods. Note that, for Themida, we enable its packing and partial code revealing models. For other pure code virtualization tools, such as Code Virtualizer [56] and VMProtect [77], researchers rely

[‡]For the Intel Haswell microarchitecture CPU, it can record 16 most recent branch pairs. For the Intel Skylake microarchitecture CPU, the recording number increases to 32 [20].

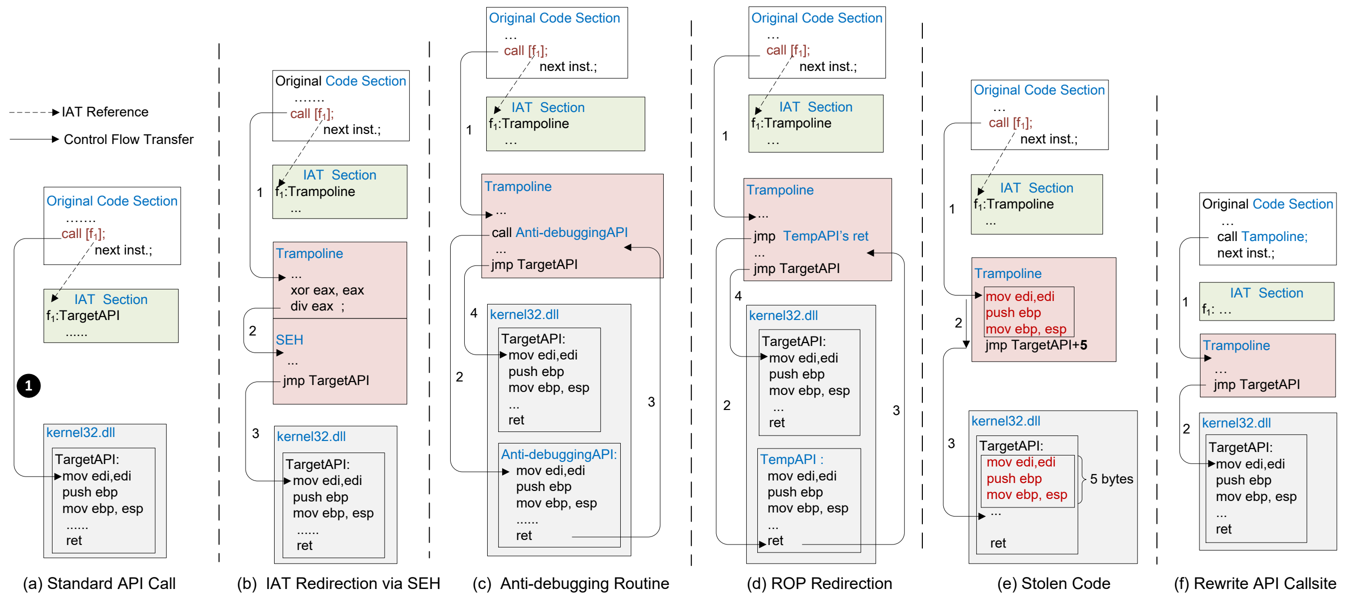


Figure 2: The examples of different API obfuscation schemes (Figure 2(b)∼ Figure 2(f)). The unpacking routine allocates and maintains a "trampoline" code area (labeled as red color boxes) to complicate the standard API call chain.

Table 3: The summary of various API obfuscation techniques. The branches involved in the complicated control flow between the original code and the target API (Column 2) could be a very large number.

Obfuscation Type	Control Flow	Cited Work
Normal API Call	Original Code ⇒ TargetAPI	—
IAT Redirection	Original Code ⇒ Trampoline ⇒ TargetAPI	[43, 63, 70]
Rewrite API Callsite	Original Code ⇒ Trampoline ⇒ TargetAPI	[63]
Anti-debugging Routine	Original Code ⇒ Trampoline ⇒ Anti-debugging API ⇒ Trampoline ⇒ TargetAPI	API-Xray
ROP Redirection	Original Code ⇒ Trampoline ⇒ End of TempAPI ⇒ Trampoline ⇒ TargetAPI	API-Xray
Stolen Code	Original Code ⇒ Trampoline ⇒ TargetAPI+n	[41, 43, 63, 70]

on totally different approaches [19, 68, 84, 85] to recover virtualization protected code, and therefore they are out of our scope.

Existing import table reconstruction approaches commonly rely on a number of assumptions that may not reflect the complexity of advanced packers. In particular, these assumptions include: 1) the address of a target API is statically identifiable in the unpacked code [2, 15, 69]; 2) when the control flow arrives at a DLL, it necessarily points to the target API's entry point [40, 62, 73]; 3) API calls have to be forwarded through the IAT [2, 15, 40, 62, 69, 73]. We conduct our study with the following three questions in mind. Unfortunately, our deep inspection gives negative answers to all of them.

Q1: Can target APIs' addresses be statically identifiable in the unpacked code?

Some methods use memory static scanning to reconstruct import tables [2, 15, 69]. They have a simple assumption that the addresses of target APIs are statically identifiable in the OEP memory (e.g., Figure 2(a)). However, this assumption can be violated by a dynamically computed address. Figure 2(b) illustrates a complicated example of IAT redirection, which is adopted by Obsidium packer. The IAT entry points to a "trampoline" area first. This code area is maintained by the unpacking routine as the relay to obfuscate the control flow ❶ in Figure 2(a). For Figure 2(b), the trampoline further installs a custom structured exception handler (SEH) and intentionally executes an erroneous instruction (e.g., division by zero) to jump to the SEH at another place. Finally, the SEH forwards the control flow to the target API. Without executing the trampoline code and SEH in Figure 2(b), we cannot identify the target API address.

Q2: When the control flow arrives at a DLL, does it necessarily point to the target API's entry point?

Dynamic-based approaches hold a common assumption that if the control flow reaches a DLL, it necessarily points to the target API's entry point. However, we find a few counterexamples that defy this assumption, and we summarize them into three types. **First**, we find some packers (Armadillo, PEP, and Obsidium) call the anti-debugging APIs before the target API (as shown in Figure 2(c)). These anti-debugging APIs perform timing checks or checksum for the anti-analysis purpose. **Second**, some packers (e.g., PELock and Obsidium)

use the ROP style to redirect their API calls (as shown in Figure 2(d)). That is, the trampoline first transfers the control flow to the `ret`-like instruction of a temporary API; then the control flow will go back to the trampoline again. After that, the trampoline finally forwards the control flow to the target API. Since this process, such as Figure 2(d), is similar to the ROP attack, we name it as "ROP redirection."

The **third** type is the so-called "stolen code" [15, 42, 63]. As shown in Figure 2(e), the stolen code invokes an API by first executing a few bytes copied from the head of API, and then it jumps back to the target API code right after the copied instructions. Because many API monitoring tools set hooks at the entry of an API, stolen code can evade these monitoring tools. We observe the adoption of stolen code in the packers such as Themida, PELock, and Enigma. Regarding how many bytes the stolen code can copy, our large-scale evaluation shows that it typically steals the first 3 bytes, 5 bytes, ..., until one basic-block size from the target API [63]. The goal of such a choice is to be compatible with a common design in DLLs—Position Independent Code. Otherwise, copying more bytes to the trampoline area may also include relative-addressing instructions, which can lead to an execution crash.

Q3: Are API calls necessarily referred to the IAT?

All of the existing import table reconstruction approaches assume that API calls must be referred to the IAT first. However, some packers (e.g., PEP, ASProtect, and Themida) use a direct call instruction to invoke a target API, without passing through the IAT. Figure 2(f) illustrates the high-level idea of this mechanism. To achieve this goal, these packers have to rewrite the original instruction at the API callsite. Suppose the original API call is an indirect call (machine code: FF15), these packers rewrite it as a direct call (machine code: E8). Note that the direct call instruction is one byte shorter than the indirect call, and thus these packers also add one padding byte to the direct call instruction.

#Branches. We summarize the control flow transfer information of various API obfuscation techniques in Table 3. To the best of our knowledge, no previous work discussed "Anti-debugging Routine" and "ROP Redirection" ever before in the context of API obfuscation. Note that the number of branches involved in the complicated control flow, as shown in the second column of Table 3, could be very large. The maximum number encountered so far is $39,322$!

4 System Design and Implementation

4.1 Overview

The overview of API-Xray is shown in Figure 3. The input to API-Xray is the OEP memory captured by a binary unpacking tool (❶ in Figure 3), such as PinDemonium [50], CAPE [18], or BinUnpack [15]. At this moment, the unpacking routine has finished multi-layer unpacking, and the control flow just jumps back to the malware's OEP. Then, the binary unpacking tool imports API-Xray as a custom DLL to reconstruct import tables (❹), which are finally stitched together with the unpacked code to assemble an executable malware sample for further analysis. API-Xray's memory static analysis module explores all possible API callsites in the OEP memory (❷); then API-Xray enforces the execution at each API callsite to efficiently pass through the trampoline code (❸). At the same time, the underlying hardware tracing offers a transparent environment to capture the branch that jumps to a DLL's memory page. Note that this branch may not point to the target API. We use the heuristics of trampoline address scope collected from hardware tracing to further determine whether the current branch reaches the target API. Compared with the existing work, API-Xray's static and dynamic analyses (❷ & ❸) work in concert to amplify each other's benefit.

4.2 Memory Static Analysis

When the packed malware's OEP is reached, we first attach the Windows Debugger (WinDbg) to the packed malware process. Then we use IDA Pro to disassemble the OEP memory via IDA WinDbg plugin [36]. After that, we run our custom IDA Pro plugin, which follows Eureka's search algorithm [69], to explore all potential API callsites. We first locate all indirect call and jump instructions; then we rule out the following cases: 1) control flow instructions whose targets reside within the unpacked program; 2) indirect jumps that access a lookup table for switch-case handlers; 3) valid API callsites through standard IAT reference. The remaining `CALL` or `JMP` instructions that have unrecognized targets are potential API callsites. After that, we save the disassembly code and detach WinDbg. The reason for doing so is that during our API Micro Execution, the trampoline code may detect the presence of a debugging environment (see Figure 2(c)).

4.3 API Micro Execution

To get rid of the complex control flow shown in the second column of Table 3 and resolve API names, we need to meet the following two requirements:

1. **Req1**: executing the trampoline code associated with each API callsite, so that we can efficiently pass through lengthy, back-and-forth jumps;

2. **Req2**: capturing the control flow branch whose destination eventually resides within the target API's code.

As shown in Figure 2's red-colored boxes, the trampoline code contains various types to obfuscate API name resolution, such as SEH, junk code, call stack preparation for running anti-debugging APIs, the code for control flow relay, and a few bytes of stolen code. Our key observation is, given the runtime context of OEP memory, *the trampoline code can run*

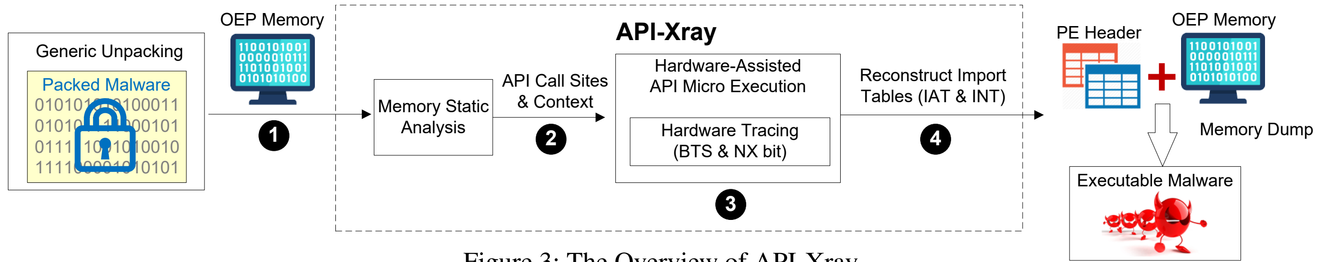


Figure 3: The Overview of API-Xray.

independently. Recall that binary packers are directly applied to the original binary code. The attached unpacking routine for the packed program is unaware of the original code logic. It is also the unpacking routine to allocate and maintain the trampoline code area. Therefore, the trampoline code's execution does not depend on the particular API arguments of the original code. Ugarte-Pedrero et al.'s longitudinal study in S&P'15 [73] also confirms that the trampoline code's execution is independent of the original code.

To meet **Req1**, our API Micro Execution creates a new thread from the address of each API callsite to dynamically execute the trampoline code. We borrow the name of "Micro Execution" from Patrice Godefroid's ICSE'14 work [33], which uses a runtime virtual machine to execute "any code fragment without a user-provided test driver or input data". Similarly, we enforce executing each API callsite without requiring concrete function arguments. When we decide that the control flow has just arrived at the target API, we terminate the current thread because we can already resolve the API name. Then we start a new API Micro Execution thread to explore the next API name. In this way, we can resolve API names one-by-one without raising any conflict. In what follows, we explore how we achieve **Req2** by taking advantage of BTS-based tracing and NX bit.

4.4 Hardware-Assisted Tracing

For the initiated API Micro Execution thread, its backend runs a hardware-assisted control flow monitoring system. Table 2 compares three hardware tracing mechanisms. LBR and IPT exhibit low runtime overhead, but they do not meet our requirements. LBR is limited by the number of branch pairs it can record (16 or 32), while IPT does not record unconditional direct branches. IPT will cause our tracing to miss the branch whose destination address just hits the target API. Therefore, we adopt BTS branch tracing and set the threshold of BTS buffer as 1000. Once BTS buffer is full, it will trigger a system interrupt, and our predefined interrupt handler will save BTS buffer's record, reset it, and resume BTS's monitoring. In this way, we do not lose any branch.

Another question is how to set up the "checkpoint", so that we can timely inspect the recorded branches. The recent work [9] takes the strategy of 1-branch interruption; that is, BTS has to be interrupted for security checking at every control flow deviation instruction. However, this design will become a performance bottleneck in our scenario. For example, Obsidium packer's trampoline code can execute up to $39,322$ branches before reaching the target API. Our solution is to enable NX bit for DLL's memory pages, and we hook page fault handler to copy recorded branch data for further inspection. As shown in Figure 4, API-Xray's implementation consists of multiple kernel-level and user-level components.

4.5 Kernel Module

API-Xray's kernel module is responsible for three main tasks. **First**, it configures and enables BTS branch tracing. **Second**, the kernel module hooks the related kernel functions that are used to enable/disable and detect NX bit. In particular, we call "ZwProtectVirtualMemory" and "ZwQueryVirtualMemory" to enable or disable NX bit for DLL and non-DLL pages. Prior to API Micro Execution, we only switch on NX bit for the loaded DLL pages in the target process. When API Micro Execution arrives at a DLL, it will trigger a page fault (❷ in Figure 4). We further discuss the reason for enabling/disabling NX for non-DLL pages in §4.7. The benefit of our design is that we can intercept user-level malware's manipulation to NX bit. **Third**, when a non-executable interruption is triggered in DLL pages, our customized page fault handler will take the following two actions: 1) notify the user module to save the current stack frame (❸ in Figure 4), which will be used in §4.7 to verify whether the current branch points to the target API; 2) copy BTS trace buffer to the user space for further analysis and then reset BTS buffer (❹ in Figure 4).

4.6 Process Filtering

The limitation of BTS tracing is that it is not process-specific. In addition to the target process's branch data, BTS buffer may contain the branch data coming from other processes. As we have the OEP memory, we know the memory range of the current process, including loaded DLLs. Besides, we also disassembled the OEP memory so that we know the instruction at the source address of a BTS record. To filter out the noise caused by other processes, we will search for a branch chain that meets the following three criteria: 1) for each pair (source and target) in the branch chain, both source address and destination address are within our process

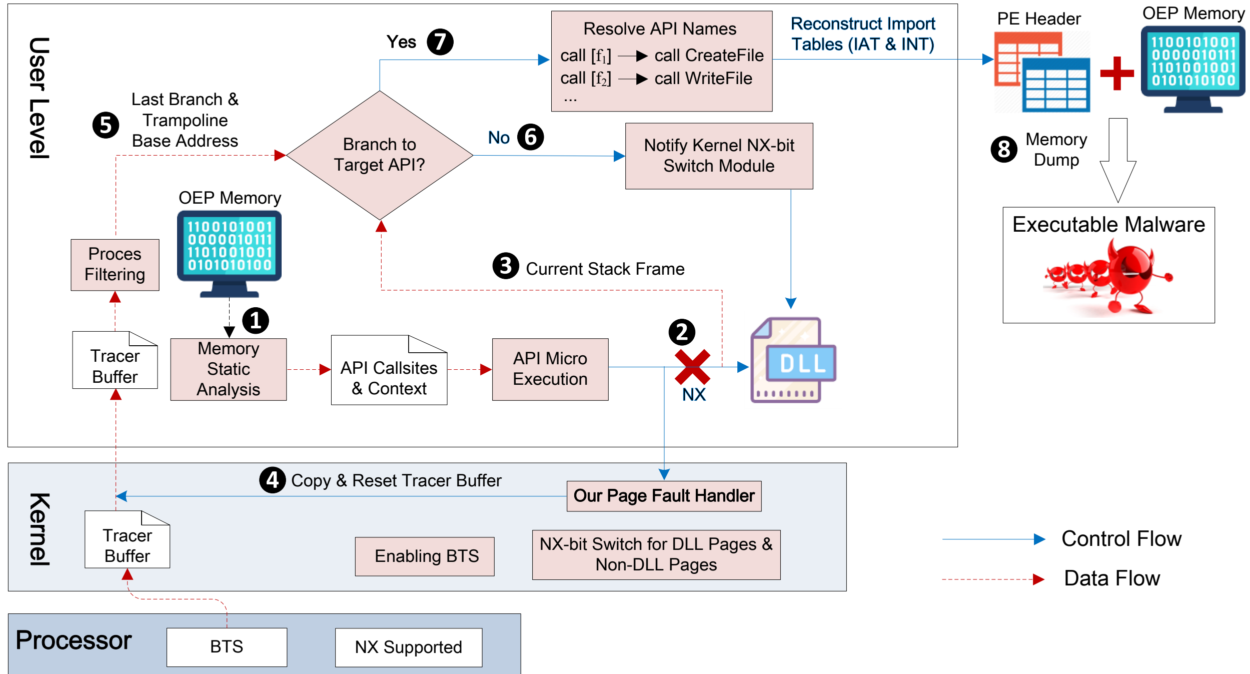


Figure 4: The detailed architecture of API-Xray. The shaded boxes in red represent API-Xray's functional modules.

memory range; 2) the instruction at the source address must be a branch instruction, because the source address from a different process may not correspond to a branch instruction in the current process; 3) the last record in the branch chain transfer the control flow from a non-DLL location (i.e., an address in the trampoline code area) to an address located within a DLL page range.

Among the collected branch chain, the first branch jumps to the trampoline code area (either from an API callsite or a DLL's memory page); the last branch jumps to a DLL's memory page. The rest of branches represents the back-and-forth jumps that occur within the trampoline code area. Based on the collection of trampoline code addresses, which share the same high bytes, we can quickly infer the base address of the trampoline code. This information will also be used in §4.7 to check a valid branch pointing to the target API. The output of our process filtering is the last valid branch, as well as the trampoline code's base address (❺ in Figure 4).

4.7 Destination Address Checking

This subsection discusses how to verify whether the destination address is located at the target API. Recall that when the control flow arrives at a DLL, it does not necessarily point to the target API. We need to manage two counterexamples: "Anti-debugging Routine" (Figure 2(c)) and "ROP Redirection" (Figure 2(d)). For the rest of the cases shown in Figure 2, they will pass our destination address checking.

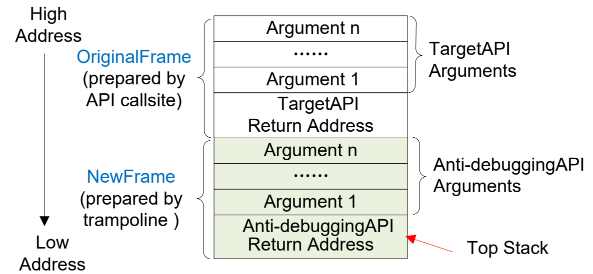


Figure 5: The stack frame of "Anti-debugging Routine."

Anti-debugging Routine. Given the destination address of a branch, we rule out the case of jumping to an anti-debugging API using: 1) its stack frame when a DLL's non-executable interruption is triggered (as shown in Figure 5), and 2) the trampoline base address. The current stack frame of this anti-debugging API is prepared by the trampoline, and the top of the stack stores the return address, which also points to the trampoline code area. If the high bytes of this return address matches the trampoline base address, we decide the current branch does not point to the target API.

ROP Redirection. To detect this counterexample, we first disassemble the instruction at the destination address; if the instruction is one of "ret-like" instructions, it means current control flow is caused by "ROP Redirection." "Ret-like" instructions include the `ret` instruction and its semantically-equivalent instructions (e.g., "pop x; jmp * x") [14].

NX-bit Switch. For the above two cases that do not point to the target API (❻ in Figure 4), we will notify the kernel

NX-bit switch module to disable NX bit for DLL pages and enable NX bit for the trampoline code. As a result, the execution of the above two cases can resume. When the control flow goes back to the trampoline, it will also trigger a non-executable page fault. At this time, our page fault handler will switch the NX bit again; that is, it switches on NX bit for DLL pages and switches off NX bit for the trampoline code. In this way, we can enable NX bit for DLL pages whenever the control flow reenters them.

1-Branch Interrupt. We have to consider the case that the control flow does not go back to the trampoline code. A skilled attacker can create ROP-chain inside the DLL to jump to the target API directly. Since we have disabled NX bit for DLL pages to resume the "ROP Redirection" execution, this attack will not trigger the page fault when the control flow reaches the target API. Our solution is to reset the size of BTS buffer as *one* when we detect "ret-like" instructions. This enables our BTS-mechanism to capture each branch in the ROP-chain inside the DLL, but at the cost of higher overhead. In this way, we ensure one of our hardware-mechanism (NX or BTS) can capture the branch to the target API in any case.

4.8 Import Table Reconstruction

If the destination address resides within the target API's code (❼ in Figure 4), the next step is to resolve the API name from this address. Note that if the packer applies the stolen code technique (Figure 2(e)), the destination address will not be the entry point of the target API, but in the middle of the API code. Therefore, we identify API name not by its entry point, but by a memory range of this API. More concretely, we first scan the OEP memory to obtain each loaded DLL's memory range. For each DLL, we scan its export address table from the DLL header to get all API names and calculate their memory ranges. After that, we relate the destination address with a particular API name by checking whether the destination address is located within the memory range of an API. After we complete all possible API Micro Executions and resolve the API name for each API callsite, we will rebuild a new IAT as well as the associated INT. Furthermore, we will recover the reference to the new IAT & INT from the PE header, so that they are reachable for static analysis and Windows PE loader. For the cases of "Rewrite Original API Call " (Figure 2(f)), we also need to rewrite direct calls back to indirect calls through the new IAT reference. At last, our recovered PE header is stitched together with the unpacked code to assemble an executable program (❽ in Figure 4).

5 Evaluation

API-Xray automates the import table reconstruction for unpacked Windows programs on the x86/x64 platform. We conduct a set of experiments to evaluate API-Xray's effectiveness from four aspects. 1) API-Xray outperforms existing work in terms of better API coverage and API-obfuscation resistance. 2) Compared with LBR and IPT, we demonstrate that our choice of BTS is the only viable option for import table reconstruction. 3) We report our experience of testing large-scale packed malware in the wild. Especially, API-Xray advances unknown/new malware detection and analysis.

Table 4: The API coverage evaluation results with the ground truth dataset. API obfuscation type numbers (Column 2) represent: 1) IAT Redirection; 2) Rewrite API Callsite; 3) Stolen Code; 4) ROP Redirection; 5) Anti-debugging Routine. We test four representative methods: BinUnpack (BU) [15], Ugarte-Pedrero et al.'s work in S&P'15 (SP) [73], RePEconstruct (RP) [44], and API-Xray (AX)).

Packers	API Obfuscation Types	#APIs			
		BU	SP	RP	AX
Non-obfuscation Packers					
UPX		348	56	348	348
API Obfuscation Packers					
Yoda's Crypter	1	102	56	124	348
Yoda's Protector	1	102	56	124	348
TELock	1	213	56	235	348
ZProtect	1	0	56	56	348
Enigma	1	23	56	59	348
ASProtect	2	178	32	202	348
PESpin	1,3	119	18	126	348
Armadillo	1,5	220	19	231	348
PEP	1,2,5	41	17	53	348
Obsidium	1,4,5	0	15	15	348
PELock	1,3,4	0	19	20	348
Themida					
Packing model	2,3	0	0	0	348
Partial code revealing	2,3	0	0	0	348

5.1 Comparative Evaluation

Our study in §3 has found that 12 prevalent packers apply different API obfuscation schemes. To set up a controlled experiment, we apply these 12 packers to a sample of notorious Zeus Trojan. Zeus Trojan, also known as Zbot, is often used to steal financial data from the victim machines and install ransomware [28]. Zeus has been on Check Point's Top10 wanted malware list for many years [71]. Our motivation for testing Zeus is based on the two following arguments. First, Zeus is the most sophisticated botnet that the FBI has ever attempted to disrupt [30]. It has 348 APIs, which is significantly more than other typical malware samples (e.g., about 114 APIs for WannaCry and about 168 APIs for Conficker). Second, Zeus is controlled by different commands from the Network, which means it has many execution paths. These execution paths cause the dynamic-based import table reconstruction approaches to recover limited APIs (see Section 2.2). In our evaluation, we compile Zeus binary code[§] with its source code in Windows 10.

We compare API-Xray with three representative import table reconstruction methods: BinUnpack [15], Ugarte-Pedrero

[§]MD5: 9e722f9c2e344f683b5e9c37b1035b95

et al.'s work [73], and RePEconstruct [44]. As we summarized in Table 1, these three methods represent memory static analysis, dynamic, and hybrid analysis, respectively. Besides, we also need a generic unpacking tool to provide the OEP memory as the input to these import table reconstruction methods. Due to the high performance of BinUnpack [15], we use BinUnpack's OEP identification heuristics to halt the process when the OEP is reached. Our testbed is a laptop with an Intel Core i7-8550 processor (quad-core, 1.80GHz) and 16GB memory, running Windows 10.

5.1.1 API Coverage

The 12 packers that apply API obfuscation are shown in the first column of Table 4. In addition, we use UPX packer to represent the packer that does not apply any obfuscation. Themida [57] is a sophisticated commercial code obfuscator. We use Themida to evaluate two complicated packer cases. **First**, we enable Themida's packing model to pack Zeus's binary code. The distinct feature of Themida packer is that the unpacking routine code is further obfuscated by code virtualization. **Second**, as the source code of Zeus is available, we apply Themida's optional functionality: "Encode Macro". "Encode Macro" allows users to mark a region of source code that needs to be encrypted. At run time, Themida will first decrypt the code inside the macro, execute it, and then encrypt it again. We treat this "Encode Macro" model as the partial code revealing packer, which is a well-known challenge for all generic unpackers [7], because only a portion of the original code is revealed during any given unpacking time window.

We enable Themida's "Encode Macro" option to protect the major functions of Zeus. This means every time only one function's OEP memory is available for us to analyze. We handle this tough case using the following steps: 1) when the unpacking tool returns the OEP memory for each function, we resolve API names for this function and dump this function's process memory; 2) we collect all resolved API names to reconstruct import tables; 3) we reassemble all function process memory dumps as a single consistent code image, which is further stitched together with reconstructed import tables from step 2 to generate an executable Zeus.

Table 4's second column shows the API obfuscation types adopted by these packers. We can see that the "IAT Redirection" is the most common API obfuscation type. Columns 3∼6 show the number of APIs that are restored by the four tested methods. We treat this number as the metrics to measure the completeness of import table reconstruction tools.

The original Zeus has 348 APIs in its import table. As a static-only method, BinUnpack [15] is brittle when handing API obfuscation schemes, so BinUnpack fails to resolve API names for ZProtect, Obsidium, PELock, Themida. For the rest of packers, we notice that BinUnpack can resolve part of API names. We look into these packers and find that these packers only obfuscate APIs exported from particular DLLs.

Table 5: The comparison of API-obfuscation resistance. "●" means this tool can defeat an API obfuscation type.

Obfuscation Type	BinUnpack	S&P'15	RePEconstruct	API-Xray
IAT Redirection		●	●	●
Rewrite API Callsite		●	●	●
Stolen Code				●
ROP Redirection				●
Anti-debugging Routine				●

For example, Yoda's Crypter packer only obfuscates the APIs exported by kernel32.dll, user32.dll, and advapi32.dll, but not other APIs. As a result, BinUnpack can restore 102 APIs that are not obfuscated by Yoda's Crypter packer.

For Ugarte-Pedrero et al.'s work [73], as a dynamic-only method, it can resolve at most 56 API names for nine packers, because only these 56 APIs are called during a single execution path. Of course, we can expect Ugarte-Pedrero et al.'s work to cover more APIs after it explores more paths with new inputs, but its design does not deal with all API obfuscation types. For the left four complex packers, Ugarte-Pedrero et al.'s work performed even worse because it is evaded by "Stolen Code," "ROP Redirection," and "Anti-debugging Routine." For RePEconstruct [44], although it can cover more APIs for some packers due to its hybrid analysis style, we can see a precipitous decline for the advanced packers that incorporate multiple anti-analysis methods. In contrast, API-Xray succeeds in resolving API names for all tested packers.

5.1.2 Resistance Against API Obfuscation Schemes

Next, we zoom in on the resistance against different API obfuscation schemes, and the results are shown in Table 5. BinUnpack's advantage lies in quickly determining the end of unpacking, but its API name resolution function is weak. Our evaluation shows that BinUnpack does not handle any API obfuscation schemes. For Ugarte-Pedrero et al.'s work and RePEconstruct, their dynamic analysis style can naturally defeat "IAT Redirection" and "Rewrite API Callsite." However, they can be cheated by the "Stolen Code" as well as two obfuscation schemes that we first unveil in this paper: "ROP Redirection" and "Anti-debugging Routine." They capture the API that they first encounter in a loaded DLL instead of the real API. API-Xray makes a clean sweep in the API obfuscation resistance comparison.

5.2 LBR vs. IPT vs. BTS

Since for modern Intel processors, there are three mechanisms to trace branch instructions, including Last Branch Record (LBR), Branch Trace Store (BTS), and Intel Processor Trace (IPT) (see §2.3). We conduct a separate experiment

to compare BTS with another two similar hardware tracing mechanisms: LBR and IPT.

Table 6 shows our evaluation results. Column 1 lists all of the packers we tested. Column 2 shows the maximum number of control flow deviation instructions from an API callsite to its target API. We can see that some numbers have already exceeded the limit of LBR stack size (16 or 32), and the peak value (39,322) comes from Obsidium packer. Column 3 presents the last branch instruction to the target API. Note that some packers use "**jmp/call immediate**" addressing (the instruction in bold) to branch to the target API, but IPT does not record these instructions. Column 4∼6 presents the running time when API-Xray adopts LBR, IPT, and BTS to monitor control flow, respectively. For each version, we report two overhead numbers. The first number represents the running time of hardware tracing mechanism, and the second one is the total running time for import table reconstruction. The blank value means this version fails to restore a complete import table. Note that IPT's running time includes both online logging and expensive offline decoding. The overhead of BTS-based version is between LBR-based and IPT-based versions, but only BTS-based version succeeds in all cases.

The LBR-based version cannot restore a complete import table if the "Maximum Branch Times" exceeds 32, and the IPT-based version fails if the "Last Branch Instruction" is a direct unconditional jump (e.g., **jmp/call immediate**). Since BTS provides a complete branch tracing capability that cannot be offered by LBR or IPT, we use BTS as our branch tracing mechanism to defeat API obfuscation. Considering that API-Xray frees security analysts from the burden of manually rebuilding import tables, its overhead is moderate.

5.3 Large-Scale Evaluation with Packed Malware In the Wild

From July 2019 to December 2019, API-Xray has been deployed into an anti-malware company for large-scale evaluation with packed malware in the wild. API-Xray is integrated into a commercial unpacking tool to assist security professionals in malware offline analysis. We have collected a total of 341,269 packed malware binaries in the production environment. 74.6% of them are protected by known packers, and the other (25.4%) are protected by custom packers.

5.3.1 API Obfuscation Distribution

Table 7 shows the distribution of various API obfuscation types in our large-scale dataset. Similar to our observation in Table 4, the "IAT Redirection" is the most popular API obfuscation type (36.5%). The type of "ROP Redirection" only accounts for less than 7% due to its high development cost. Considering a packer can combine different API obfuscation types, we also count the number of API obfuscation schemes used in a packer. As shown in Figure 6, 48.1% of packed samples do not apply any API obfuscation scheme. The remaining 51.9% of them (total 177,119) apply at least one API obfuscation type, and 2.6% of packed samples apply the maximum four API obfuscation types.

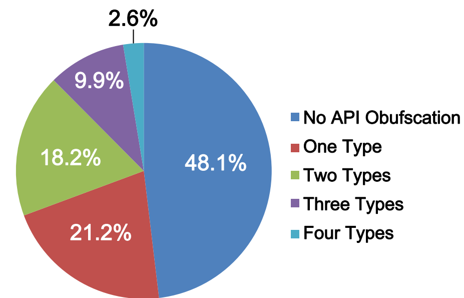


Figure 6: The statistics of API obfuscation types used by packed malware in the wild.

5.3.2 Evaluation Results

For these 177,119 packed malware samples that are also protected by API obfuscation schemes, we apply API-Xray to their OEP memory to reconstruct import tables. Figure 7 presents the cumulative distribution of our analysis results. We first count the number of branches recorded by API-Xray, as this number reflects the complexity of the control flow between an API callsite and its target API. As shown in Figure 7(a), about 29.7% of samples (total 52,604) generate more than 32 branches, which exceed the size of LBR stack.

However, for these packed malware samples, we do not have their source code or the binary code with no packer applied as a reference. To evaluate whether API-Xray reconstructs import tables successfully, we use two heuristics.

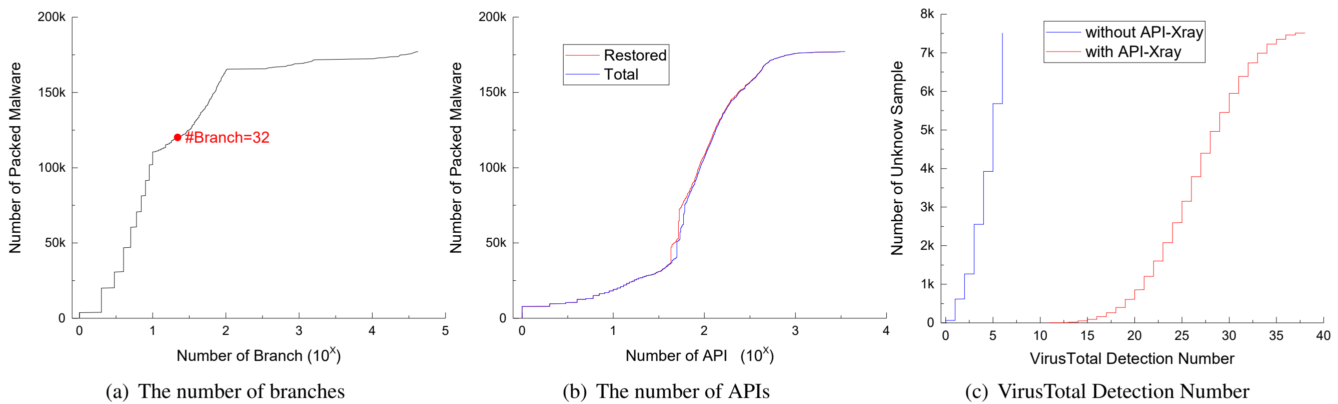
Heuristics 1. We use our custom IDA Pro plugin to scan API-Xray's outputs to check whether there exists an API call with an unresolved name. If yes, we consider this sample has an incomplete import table. In our evaluation, we find that API-Xray succeeds for 98.4% of samples (total 174,285). We investigate the remaining 1.6% of samples and find out that these samples call some APIs exported from custom DLLs, but they are absent in the our testing environment. Figure 7(b) shows the number of APIs restored by API-Xray and the number of total APIs, respectively. The two lines in Figure 7(b) are very close to each other, which means API-Xray only misses a very small portion of custom APIs.

Heuristics 2. We also evaluate the executability of API-Xray's outputs. We run each unpacked PE file in three state-of-the-art malware sandboxes: SecondWrite [67], Hybrid Analysis [22], and VMRay Analyzer [78]. We select them for two reasons: 1) they all report whether a sample is malicious or not; 2) since these three sandboxes apply different anti-evasion methods, a malware sample is possible to evade one of them but hard to evade all of them. As shown in Figure 8,

Table 6: The comparison of three hardware tracing mechanisms.

Packers	#Max-Branches	Last Branch	Running Time (seconds)		
			LBR	IPT	BTS
Yoda's Crypter	14	**jmp imm**[1]	(0.11, 13.2)		(7.3, 16.7)
Yoda's Protector	10	**jmp imm**	(0.11, 13.8)		(7.2, 17.2)
TELock	14	ret	(0.12, 9.4)	(39.4, 52.6)	(7.8, 17.1)
ZProtect	10	ret	(0.11, 13.4)	(37.0, 50.3)	(7.4, 17.0)
ASProtect	**45**	call eax		(43.1, 55.4)	(8.6, 16.6)
PESpin	13	**jmp imm**	(0.13, 13.6)		(8.5,17.7)
Armadillo	28	call dword []	(0.13, 10.8)	(43.4, 54.1)	(8.6, 15.1)
Enigma	12	**call imm**[1]	(0.13, 14.6)		(10.1, 19.5)
PEP	13	**jmp imm**	(0.13, 11.0)		(5.3, 13.5)
Themida	**60**	**jmp imm**			(9.2, 18.6)
Obsidium	**39,322**	ret/call ecx		(130.8, 161.4)	(26.1, 43.6)
PELock	**92**	**call imm**			(14.4, 25.6)

[1] e.g., jmp 0x73dc17c8 and call 0x73dc17c8



(a) The number of branches (b) The number of APIs (c) VirusTotal Detection Number

Figure 7: The cumulative distribution results of evaluating large-scale packed malware in the wild.

Table 7: The distribution of API obfuscation types.

API Obfuscation Type	Distribution
Type 1: IAT Redirection	36.5%
Type 2: Stolen Code	12.7%
Type 3: Rewrite API callsite	11.8%
Type 4: Anti-debugging Routine	7.8%
Type 5: ROP Redirection	6.9%

84.4% of unpacked PE files (total 149,488) are labeled as "Malicious" by at least one of the three sandboxes.

Two Evaluation Heuristics Comparison. Compared with the results calculated by Heuristics 1, we know that 24,797 unpacked PE files do not exhibit malicious behaviors in any sandbox, even they have complete import tables. Upon further investigation, we categorize them into three classes.

First, we find 16,849 samples crashed at run time. For these samples, we utilize a "Just-In-Time" debugger [26] to capture the crash address automatically. We find that the crash occurs at the address around the original entry point (OEP) but not at any API callsite. It indicates that the unpacking tool does not accurately identify the unpacked programs' OEPs. The root cause is that some custom packers apply heavyweight code obfuscation around the OEP area to undermine the existing OEP search heuristics. We leave addressing this problem as our future work. Second, we find 7,789 samples are trigger-based malware. They do not perform any malicious actions because the trigger condition is not met (e.g., revealing malicious behavior on a particular date). Finally, 159 unpacked malware samples are able to detect all of the three sandboxes and then hide their malicious behaviors.

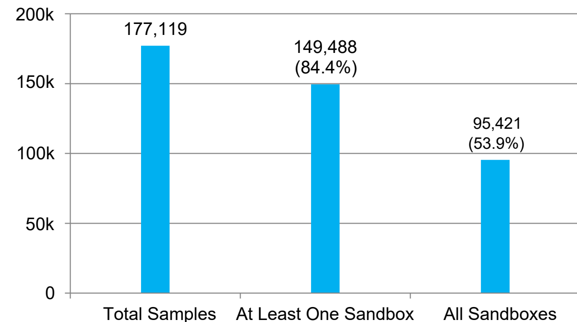


Figure 8: The number of recovered PE files exhibiting malicious behaviors in three different sandboxes.

VirusTotal Detection Number. Security analysts also find that, without applying API-Xray, 7,514 pieces of unpacked malware are not well recognized by anti-virus scanners. We treat these 7,514 samples as unknown or new malware

Table 8: The case study of an unknown malware sample.

Sample	#APIs		#VirusTotal	
	Unpacked Code	API-Xray	Unpacked Code	API-Xray
Unknown Trojan[1]	0	63	2	33

[1] MD5: d4f377c849b86d5ca89776bc56eea832.

because they meet the following two criteria: 1) less than 10% of anti-virus scanners from VirusTotal [76] label them as malware; 2) if they have malware labels, the labels are either "Generic" or "Heuristic", rather than a specific malware name (e.g.,"Zeus" or "WannaCry"). Figure 7(c) shows the VirusTotal detection numbers for these unknown malware before/after applying API-Xray. As API-Xray recovers the metadata of imported APIs that can provide valuable insight into the malicious intention, 8 to 32 additional anti-virus scanners (the average number is 22) are able to recognize the unknown malware samples.

We take an unknown Trojan sample as an example to demonstrate that API-Xray improves the accuracy of unknown malware detection. This sample stealthily downloads other malicious files from a remote server, and then it installs and executes the files. It uses "IAT Redirection" and "Anti-debugging Routine" to hide API names, such as "InternetReadFile" and "WinExec". The malicious behavior of this sample hinges on the invocation of particular APIs, but its binary code exhibits no recognizable signatures, such as unique strings or byte n-grams. Table 8 shows without the API information, only two anti-virus scanners recognize this malware's unpacked code. After API-Xray recovers the 63 APIs of this sample, the detection number of VirusTotal raises to 33.

6 Discussion

A perfect malware analysis solution is unattainable. The cyber arms race between malware and defenders has transformed into an intensive tug-of-war. Cybercriminals are motivated to circumvent API-Xray once it is public. We do not assume that evading API-Xray is strictly impossible, but it can prohibitively increase malware developers' cost. This section discusses possible attacks to API-Xray, our countermeasures, API-Xray's limitations, and the application to Linux malware.

6.1 Possible Attacks and Countermeasures

Attacks to BTS. The BTS mechanism can only be manipulated in the kernel. Starting with Windows 10 (version 1607), Windows OS does not load any new kernel drivers unless they are signed by Windows Hardware Dev Center program [13]. This mandatory driver signing enforcement leaves malware with little wiggle room to hack into the OS kernel.

Attacks to NX bit. Unlike BTS mechanism, the NX bit can be detected and manipulated at the user level. API-Xray's kernel module can intercept the detection and manipulation from user-mode malware samples and deceive them by returning expected answers. For example, malware can use API "VirtualQuery" to detect whether API-Xray has enabled the NX bit for DLLs' virtual memory pages. However, we also hook its corresponding native API "ZwQueryVirtualMemory", in which we modify the return value to hide the NX bit. Similarly, malware can call the API "VirtualProtect" or "VirtualAlloc" to disable the NX bit [58]. This attack is prevalent in ROP attacks [12, 59, 61, 66]. However, the VirtualProtect and VirtualAlloc will call the related Windows native API eventually: "ZwProtectVirtualMemory" and "ZwAllocateVirtualMemory". To prevent this attack, we have hooked both "ZwProtectVirtualMemory" and "ZwProtectVirtualMemory" in our current design. Since disabling NX from the user level can only be accomplished via "VirtualProtect" and "VirtualAlloc" [80], our kernel-level hooking will protect API-Xray from this attack. We have evaluated the detection and manipulation attempts to the NX bit in the userspace. As shown in Table 9, the API-Xray's kernel module can defeat the attacks to NX bit successfully.

Table 9: Detection & prevention to NX-bit attacks.

Attack Type	Countermeasure	Result
Detect NX		
VirtualQuery	ZwQueryVirtualMemory	✓
Disable NX		
VirtualProtect	ZwProtectVirtualMemory	✓
VirtualAlloc	ZwAllocateVirtualMemory	✓

Statically-Linked Library. If system libraries are statically linked into malware binary code, API-Xray cannot resolve API names because malware will never call APIs from our monitoring system's DLLs. However, we argue that static linking is not an attractive option to spread malware. First, it causes incompatibility problems under different Windows versions. Second, static linking also compromises malware's portability, because it bloats program size drastically.

Stolen Function. Kawakoya et al. [41] describe an evolved version of stolen code: instead of copying a few bytes from the head of an API, it copies the whole body of an API. We call it as "stolen function." API-Xray will miss this case because the control flow does not jump to the target API at all. However, it is not a trivial task to copy all instructions of an API to another memory space and then execute them smoothly. The stolen function has to relocate all related position-dependent code in advance; otherwise, it will lead to an execution crash. To counter the stolen function, we can leverage the "Execute-no-Read" idea [5] to protect the DLL memory pages as "no-Read." When the target API function is copied to a new location, it will be monitored by our page fault handler. And then, we use the target API information (name & address) to reconstruct import tables.

Argument-Sensitive Trampoline. The basic premise of our API Micro Execution is that the trampoline code does not depend on the particular API arguments. A determined packer author can customize the trampoline code for each

Table 10: Running time (seconds) of fake-API-call DoS attacks. They have relatively small impact on API-Xray.

Sample	API-Xray (s)		Relative Slowdown
	Disable	Enable	
Yoda's Protector	0.7	17.2	23.6X
Yoda's Protector + (Fake API Call)$\times 10^3$	0.8	17.3	20.6X
Yoda's Protector + (Fake API Call)$\times 10^6$	121.0	153.1	26.5%
Yoda's Protector + (Fake API Call)$\times 10^9$	121,472	123,648	1.8%

API callsite. For example, only when the trampoline code checks the validity of API arguments (e.g., a specific string or HANDLE value), it transfers the control flow to the target API. In this case, we have to resort to expensive symbolic execution to explore a feasible path to the target API.

Fake API Calls. An intuitive attack to any API-monitoring based security measures is the so-called "Fake API Calls" [15]. The packer can invoke many iterations of fake or null API calls before calling the target API. This will increase API-Xray's overhead because we have to check the destination address for every fake API call. However, API invocations are much expensive as well. BinUnpack [15] has quantitatively measured the adverse impact of fake API calls and concluded that too many fake API calls impose dramatically large overhead to the packed malware itself. Inspired by BinUnpack [15], we also simulated a fake-API-call Denial-of-Service (DoS) attack by modifying the open-source Yoda's Protector packer [23]. As shown in Table 10, the API-Xray's overall running time does not increase significantly when the fake API call iterations are less than 10^6. When the iteration number reaches 10^9 times, the custom Yoda's Protector packer's execution will be occupied by the large number of API invocations, and the runtime overhead will increase by five orders of magnitude; while API-Xray only incur 1.8% relative slowdown to the custom Yoda's Protector packer. Clearly, the accumulative overhead from a plethora of fake API calls far outstrips the deterioration of API-Xray's performance.

6.2 Limitations

API-Xray fails to produce an executable PE file from the unpacked code for the following two cases.

Custom DLLs. We find that 1.6% of malware samples call APIs exported from custom DLLs instead of standard Windows DLLs. Unfortunately, API-Xray cannot restore import tables exported from custom DLLs, which are absent in our testing environment.

OEP Obfuscation. 9.5% of unpacked PE files with complete import tables crashed at run time. The reason is OEP obfuscation schemes cause existing generic unpacking tools to miss the real OEP locations. For example, many unpacking tools use the "stack balance" detects OEP by checking whether the stack is similar to that when a program is just loaded into memory. However, some custom packers do not satisfy this rule. Dealing with OEP obfuscation is an orthogonal question to API-Xray, and we leave it as our future work.

6.3 Application to Linux Malware

API-Xray is designed to work on Intel CPUs, and both Windows and Linux OS provide the interface to manipulate BTS and NX bit. Besides, Linux's executable file format also has a similar import table structure. According to Cozzi et al.'s study [21], Linux malware is not as complex as Windows malware. Most packed Linux malware samples are protected by UPX packer or UPX-like variants, which do not apply any API obfuscation scheme. If new Linux packed malware becomes as complex as its Windows counterpart, API-Xray's technique is generalized to Linux malware as well.

7 Conclusion

API-Xray is the first hardware-assisted solution towards bridging the gap of generic binary unpacking—automated import table reconstruction. API-Xray complements the state-of-art binary unpacking tools by producing a standard PE file that can be executed and analyzed independently. Security analysts utilizing API-Xray will enjoy a simpler and more streamlined malware analysis than ever before.

Acknowledgments

We sincerely thank Usenix Security 2021 anonymous reviewers for their insightful and helpful comments. Jiang Ming was supported by the National Science Foundation (NSF) under grant CNS-1850434. We thank the University of Texas at Arlington and the Department of Education for supporting us with a Graduate Assistance in Areas of National Need (GAANN) fellowship. This work was also supported in part by the National Natural Science Foundation of China grants (61972297, U1636107, and 61976085). This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 830927.

References

[1] Hojjat Aghakhani, Fabio Gritti, Francesco Mecca, Martina Lindorfer, Stefano Ortolani, Davide Balzarotti, Giovanni Vigna, and Christopher Kruegel. When Malware is Packin' Heat; Limits of Machine Learning Classifiers Based on Static Analysis Features. In *Proceedings of the 27th Network and Distributed System Security Symposium (NDSS'20)*, 2020.

[2] Aguila. Scylla - x64/x86 Imports Reconstruction. https://github.com/NtQuery/Scylla, 2016.

[3] Mansour Ahmadi, Dmitry Ulyanov, Stanislav Semenov, Mikhail Trofimov, and Giorgio Giacinto. Novel Feature Extraction, Selection and Fusion for Effective Malware Family Classification. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY'16)*, 2016.

[4] Pieter Arntz. Analyzing Malware by API Calls. http://tiny.cc/qm6rsz, Malwarebytes Labs Blog, October 2017.

[5] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*, 2014.

[6] M. Bazús, R.J. Rodríguez, and J. Merseguer. Qualitative and Quantitative Evaluation of Software Packers. NoConName 2015, 2015.

[7] Leyla Bilge, Andrea Lanzi, and Davide Balzarotti. Thwarting Real-time Dynamic Unpacking. In *Proceedings of the Fourth European Workshop on System Security (EUROSEC'11)*, 2011.

[8] Guillaume Bonfante, Jose Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier, and Aurélien Thierry. CoDisasm: Medium Scale Concatic Disassembly of Self-Modifying Binaries with Overlapping Instructions. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[9] Marcus Botacin, Paulo Lício De Geus, and André Grégio. Enhancing Branch Monitoring for Security Purposes: From Control Flow Integrity to Malware Analysis and Debugging. *ACM Transactions on Privacy and Security (TOPS)*, 21(1):4, 2018.

[10] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent Dynamic Instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE'12)*, 2012.

[11] Joan Calvet, Fanny Lalonde Lévesque, Jose M. Fernandez, Erwann Traourouder, Francois Menet, and Jean-Yves Marion. WaveAtlas: Surfing Through the Landscape of Current Malware Packers. Virus Bulletin Conference, 2015.

[12] Nicholas Carlini and David Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *23rd USENIX Security Symposium (USENIX Security'14)*, pages 385–399, 2014.

[13] Microsoft Hardware Dev Center. Driver Signing Policy. http://tiny.cc/dm6rsz, April 2017.

[14] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-Oriented Programming without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, 2010.

[15] Binlin Cheng, Jiang Ming, Jianmin Fu, Guojun Peng, Ting Chen, Xiaosong Zhang, and Jean-Yves Marion. Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS'18)*, 2018.

[16] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert H Deng. ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks. In *In Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[17] Seokwoo Choi. API Deobfuscator: Identifying Runtime-obfuscated API Calls via Memory Access Analysis. Black Hat Asia, 2015.

[18] Context Information Security. CAPE: Malware Configuration And Payload Extraction. https://cape.contextis.com/, 2016.

[19] Kevin Coogan, Gen Lu, and Saumya Debray. Deobfuscation of Virtualization-Obfuscated Software: A Semantics-Based Approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*, 2011.

[20] Intel Corparation. Intel ((R)) 64 and IA-32 Architectures Software Developer's Manual. *Combined Volumes, Dec*, 2016.

[21] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. Understanding Linux Malware. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P'18)*, 2018.

[22] crowdstrike. Hybrid Analysis. https://www.hybrid-analysis.com/, [online].

[23] Ashkbiz Danehkar. Yoda's Protector. https://sourceforge.net/projects/yodap/, 2013.

[24] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *23rd USENIX Security Symposium (USENIX Security'14)*, pages 401–416, 2014.

[25] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. SoK: Using Dynamic Binary Instrumentation for Security (And How You May Get Caught Red Handed). In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (AsiaCCS'19)*, 2019.

[26] Visual Studio Docs. Debug using the Just-In-Time Debugger in Visual Studio. http://tiny.cc/bm6rsz, September 2018.

[27] Ken Dunham and Egan Hadsell. Malcode Context of API Abuse. https://www.sans.org/reading-room/whitepapers/malicious/paper/33649, April 2011.

[28] Nicolas Falliere and Eric Chien. Zeus: King of the Bots. Symantec Security Response, 2009.

[29] Fareed Fauzi. Common Windows API in Analyzing and Reversing Windows Malware. https://fareedfauzi.github.io/notes/windows_api_in_reversing_malware/, August 2019.

[30] FBI. GameOver Zeus Botnet Disrupted Collaborative Effort Among International Partners. http://tiny.cc/6u8rsz, last reviewed, 10/1/2020.

[31] Xinyang Ge, Weidong Cui, and Trent Jaeger. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, 2017.

[32] Jeffrey Gennari. Static Identification of Program Behavior using Sequences of API Calls. http://tiny.cc/rm6rsz, CMU Software Engineering Institute Blogs, April 2016.

[33] Patrice Godefroid. Micro Execution. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, 2014.

[34] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard. In *23rd USENIX Security Symposium (USENIX Security'14)*, pages 417–432, 2014.

[35] Fanglu Guo, Peter Ferrie, and Tzi-Cker Chiueh. A Study of the Packer Problem and Its Solutions. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID'08)*, 2008.

[36] Hex-Rays. Debugging Windows Applications with IDA WinDbg Plugin. https://www.hex-rays.com/products/ida/support/tutorials/debugging_windbg.pdf, 2011.

[37] Ashkan Hosseini. Ten Process Injection Techniques: A Technical Survey of Common and Trending Process Injection Techniques. http://tiny.cc/wm6rsz, July 2017.

[38] Xin Hu, Sandeep Bhatkar, Kent Griffin, and Kang G. Shin. MutantX-S: Scalable Malware Clustering Based on Static Features. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*, 2013.

[39] AV-TEST Institute. Malware Statistics 2020: A look at Malware Trends by the Numbers. https://www.av-test.org/en/statistics/malware/, October 2020.

[40] Sebastien Josse. Secure and Advanced Unpacking using Computer Emulation. *Journal in Computer Virology*, 3(3), 2007.

[41] Yuhei Kawakoya, Makoto Iwamura, and Jun Miyoshi. Taint-assisted IAT Reconstruction against Position Obfuscation. *Journal of Information Processing*, 26:813–824, 2018.

[42] Yuhei Kawakoya, Makoto Iwamura, Eitaro Shioji, and Takeo Hariu. API Chaser: Anti-analysis Resistant Malware Analyzer. In *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'13)*, 2013.

[43] Yuhei Kawakoya, Eitaro Shioji, Yuto Otsuki, Makoto Iwamura, and Takeshi Yada. Stealth Loader: Trace-Free Program Loading for API Obfuscation. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'17)*, 2017.

[44] David Korczynski. RePEconstruct: Reconstructing Binaries with Self-modifying Code and Import Address Table Destruction. In *Proceedings of the 11th International Conference on Malicious and Unwanted Software (MALWARE'16)*, 2016.

[45] McAfee Labs. McAfee Labs Threats Report. https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-aug-2019.pdf, August 2019.

[46] Jinku Li, Xiaomeng Tong, Fengwei Zhang, and Jianfeng Ma. Fine-CFI: Fine-Grained Control-Flow Integrity for Operating System Kernels. *IEEE Transactions on Information Forensics and Security*, 13(6), 2018.

[47] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. Transparent and Efficient CFI Enforcement with Intel Processor Trace. In *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*, 2017.

[48] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*, 2005.

[49] Alessandro Mantovani, Simone Aonzo, Xabier Ugarte-Pedrero, Alessio Merlo, and Davide Balzarotti. Prevalence and Impact of Low-Entropy Packing Schemes in the Malware Ecosystem. In *Proceedings of the 27th Network and Distributed System Security Symposium (NDSS'20)*, 2020.

[50] Sebastiano Mariani, Lorenzo Fontana, Fabio Gritti, and Stefano D'Alessio. PinDemonium: A DBI-Based Generic Unpacker for Windows Executable. Black Hat USA, 2016.

[51] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. OmniUnpack: Fast, Generic, and Safe Unpacking of Malware. In *Proceedings of the 23nd Annual Computer Security Applications Conference (ACSAC'07)*, 2007.

[52] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In *Proceedings of the 26th USENIX Conference on Security Symposium (USENIX Security'17)*, 2017.

[53] NO-MERCY. Top Maliciously Used APIs. https://rstforums.com/forum/topic/95273-top-maliciously-used-apis/, 2015.

[54] Obsidium Software. Obsidium: Software Protection System. https://www.obsidium.de/, [online].

[55] Philip OKane, Sakir Sezer, and Kieran McLaughlin. Obfuscation: The Hidden Malware. *IEEE Security and Privacy*, 9(5), 2011.

[56] Oreans Technologies. Code Virtualizer: Total obfuscation against reverse engineering. http://oreans.com/codevirtualizer.php, [online].

[57] Oreans Technologies. Themida: Advanced Windows Software Protection System. https://www.oreans.com/themida.php, [online].

[58] Raghav Pande and Amit Malik. FireEye Threat Research—Angler Exploit Kit Evading EMET. https://www.fireeye.com/blog/threat-research/2016/06/angler_exploit_kite.html, June 2016.

[59] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security'13)*, pages 447–462, 2013.

[60] Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D'Alessio, Lorenzo Fontata, Fabio Gritti, and Stefano Zanero. Measuring and Defeating Anti-Instrumentation-Equipped Malware. In *Proceedings of the 14th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA'17)*, 2017.

[61] Aravind Prakash and Heng Yin. Defeating ROP Through Denial of Stack Pivot. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15)*, 2015.

[62] Jason Raber and Brian Krumheuer. QuietRIATT: Rebuilding the Import Address Table Using Hooked DLL Calls. Black Hat DC, 2009.

[63] Kevin A. Roundy and Barton P. Miller. Binary-code Obfuscations in Prevalent Packer Tools. *ACM Computing Surveys*, 46(1), 2013.

[64] Mark E Russinovich, David A Solomon, and Alex Ionescu. *Windows Internals (6th Edition)*. Microsoft Press, 2012.

[65] Ashkan Sami, Babak Yadegari, Hossein Rahimi, Naser Peiravian, Sattar Hashemi, and Ali Hamze. Malware Detection Based on Mining API Calls. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC'10)*, 2010.

[66] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. Evaluating the Effectiveness of Current Anti-ROP Defenses. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'14)*, 2014.

[67] SecondWrite. SecondWrite's Malware Deepview. https://www.secondwrite.com/, [online].

[68] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic Reverse Engineering of Malware Emulators. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P'09)*, 2009.

[69] Monirul Sharif, Vinod Yegneswaran, Hassen Saidi, Phillip Porras, and Wenke Lee. Eureka: A Framework for Enabling Static Malware Analysis. In *Proceedings of the 13th European Symposium on Research in Computer Security (ESORICS'08)*, 2008.

[70] Masaki Suenaga. A Museum of API Obfuscation on Win32. Symantec Security Response, 2009.

[71] Email Tara. Most Wanted Malware: Banking Trojans Come to the Fore Again. https://www.infosecurity-magazine.com/news/banking-trojans-come-to-the-fore/, September 2017.

[72] The Enigma Protector. Enigma Protector: A professional system for executable files licensing and protection. http://enigmaprotector.com/, [online].

[73] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. In *Proceedings of the 36th IEEE Symposium on Security & Privacy (S&P'15)*, 2015.

[74] Xabier Ugarte-Pedrero, Mariano Graziano, and Davide Balzarotti. A Close Look at a Daily Dataset of Malware Samples. *ACM Transactions on Privacy and Security*, 22(1), January 2019.

[75] Victor Van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical Context-Sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[76] VirusTotal. Free online virus, malware, and URL scanner. https://www.virustotal.com, [online].

[77] VMProtect Software. VMProtect software protection. http://vmpsoft.com, [online].

[78] VMRay. VMRay Analyzer: A Smarter, Stealthier Malware Sandbox. https://www.vmray.com/products/malware-sandbox-vmray-analyzer/, [online].

[79] Zhi Wang and Xuxian Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (S&P'10)*, 2010.

[80] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Securing Untrusted Code via Compiler-agnostic Binary Rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*, 2012.

[81] Te-En Wei, Zhi-Wei Chen, Chin-Wei Tien, Jain-Shing Wu, Hahn-Ming Lee, and Albert B Jeng. RePEF — A System for Restoring Packed Executable File for Malware Analysis. In *2011 International Conference on Machine Learning and Cybernetics*, 2011.

[82] Carsten Willems, Ralf Hund, Andreas Fobian, Dennis Felsch, Thorsten Holz, and Amit Vasudevan. Down to the Bare Metal: Using Processor Features for Binary Analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*, 2012.

[83] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. CFIMon: Detecting Violation of Control Flow Integrity using Performance Counters. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'12)*, 2012.

[84] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. VMHunt: A Verifiable Approach to Partial-Virtualized Binary Code Simplification. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS'18)*, 2018.

[85] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A Generic Approach to Automatic Deobfuscation of Executable Code. In *Proceedings of the 36th IEEE Symposium on Security & Privacy (S&P'15)*, 2015.

[86] Wei Yan, Zheng Zhang, and Nirwan Ansari. Revealing Packed Malware. *IEEE Security and Privacy*, 6(5), September 2008.

[87] Liwei Yuan, Weichao Xing, Haibo Chen, and Binyu Zang. Security Breaches as PMU Deviation: Detecting and Identifying Security Attacks Using Performance Counters. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, 2011.

[88] Pinghai Yuan, Qingkai Zeng, and Xuhua Ding. Hardware-Assisted Fine-Grained Code-Reuse Attack Detection. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'15)*, 2015.

[89] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical Control Flow Integrity & Randomization for Binary Executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (S&P'13)*, 2013.