Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost

Binlin Cheng*‡
Wuhan University & Hubei Normal University
Wuhan, Hubei 430072, China
binlincheng@163.com

Jiang Ming*†
University of Texas at Arlington
Arlington, TX 76019, USA
jiang.ming@uta.edu

Jianming Fu†‡
Wuhan University
Wuhan, Hubei 430072, China
jmfu@whu.edu.cn

Guojun Peng‡
Wuhan University
Wuhan, Hubei 430072, China
guojpeng@whu.edu.cn

Ting Chen
University of Electronic Science and Technology of China
Chengdu, Sichuan 611731, China
brokendragon@uestc.edu.cn

Xiaosong Zhang
University of Electronic Science and Technology of China
Chengdu, Sichuan 611731, China
johnsonzxs@uestc.edu.cn

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

ABSTRACT

Binary packing, encoding binary code prior to execution and decoding them at run time, is the most common obfuscation adopted by malware authors to camouflage malicious code. Especially, most packers recover the original code by going through a set of "written-then-executed" layers, which renders determining the end of the unpacking increasingly difficult. Many generic binary unpacking approaches have been proposed to extract packed binaries without the prior knowledge of packers. However, the high runtime overhead and lack of anti-analysis resistance have severely limited their adoptions. Over the past two decades, packed malware is always a veritable challenge to anti-malware landscape.

This paper revisits the long-standing binary unpacking problem from a new angle: packers consistently obfuscate the standard use of API calls. Our in-depth study on an enormous variety of Windows malware packers at present leads to a common property: malware's Import Address Table (IAT), which acts as a lookup table for dynamically linked API calls, is typically erased by packers for further obfuscation; and then unpacking routine, like a custom dynamic loader, will reconstruct IAT before original code resumes execution. During a packed malware execution, if an API is invoked through looking up a rebuilt IAT, it indicates that the original payload has been restored. This insight motivates us to design an efficient unpacking approach, called *BinUnpack*. Compared to the previous methods that suffer from multiple "written-then-executed" unpacking layers, BinUnpack is free from tedious memory access monitoring, and therefore it introduces very small runtime overhead. To defeat a variety of ever-evolving evasion tricks, we design BinUnpack's API monitor module via a novel kernel-level DLL hijacking technique. We have evaluated BinUnpack's efficacy extensively with more than 238K packed malware and multiple Windows utilities. BinUnpack's success rate is significantly better than that of existing tools with several orders of magnitude performance boost. Our study demonstrates that BinUnpack can be applied to speeding up large-scale malware analysis.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**;

KEYWORDS

Windows Malware Analysis, Generic Binary Unpacking, Import Address Table, Kernel-level DLL Hijacking

ACM Reference Format:
Binlin Cheng, Jiang Ming, Jianming Fu, Guojun Peng, Ting Chen, Xiaosong Zhang, and Jean-Yves Marion. 2018. Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost. In *CCS '18: 2018 ACM SIGSAC Conference on Computer & Communications Security Oct. 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3243734.3243771

*Both authors contributed equally to the paper.
†Corresponding authors: jiang.ming@uta.edu and jmfu@whu.edu.cn.
‡(1) Key Laboratory of Aerospace Information Security and Trust Computing;
(2) School of Cyber Science and Engineering, Wuhan University

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243771

1 INTRODUCTION

Malicious software (malware) has become a significant threat to cybersecurity. Big malware attacks such as ransomware ripple across the world and cause catastrophic damage [6, 66]. Driven by the rich profit, cyber-criminals are highly motivated to undermine malware detection/analysis by applying numerous obfuscation schemes [94]. Among them, binary packing is believed to be a panacea to thwart the widely used anti-virus scanning [11, 50, 65, 70]. Binary packers first encode malware code through encryption or compression and attach an unpacking routine to the packed malware. In addition, packers also erase the Portable Executable (PE) file's import address table (IAT) to complicate the analysis of API calls. When a packed malware starts running, the unpacking routine first decodes the payload binary to memory pages and reconstructs its IAT. After that, the execution flow will jump to the original entry point (OEP) to resume malware payload execution. In this way, the actual malicious code and data stays unrecognizable until run time, making it immune to malware analysis techniques that measure static features. For example, applying machine learning to large-scale packed malware will lead to the detection of packers rather than malicious behavior [100]. An embarrassing fact is many anti-virus scanners, which are widely deployed at the end host, take a particular packer's signature as the identification of malware [93, 104]. A study in 2013 shows that nearly 70% of packed Windows system files are falsely labeled as malware [67].

Binary packing technique has evolved from the simple, single-layer packer to the complicated, multi-layer packer with a variety of anti-analysis tricks [97]. When a packed binary starts running, the original code is *written* in memory pages sometime and then get *executed*. Besides, this "written-then-executed" procedure can iterate many times, i.e., the dynamically generated code itself may continue generating new code and executing it. Each iteration of dynamically generated code is called a *layer* [97] or *wave* [14]. The previous generic unpacking tools have fully utilized this feature by tracking the "written-then-executed" instructions [3, 9, 36, 55, 77, 81] or memory pages [28, 31, 49, 56]. However, *no silver bullet can sharply determine the end of multi-layer unpacking* because it has been proven to be an undecidable problem [83] [1]. The existing approaches have to continuously monitor all possible "written-then-executed" layers and detect the existence of original code in a certain layer with several heuristics [28, 35, 37, 55]. All of these factors contribute to the high runtime overhead imposed by current generic unpacking tools, making them too expensive for wide deployment.

The second notable feature of sophisticated packers is that they adopt different anti-analysis techniques to impede unpacking attempts [13, 21, 82]. To track a packer's self-decoding progress, generic unpacking typically relies on dynamic analysis techniques, such as debugging [17, 83, 91], dynamic binary instrumentation [3, 9, 55, 77, 81], system emulator [8, 29, 36, 97], and API hooking [31, 49, 56]. However, most of them are not transparent to the packers. Evasive packers can fingerprint these analysis systems and, as a result, terminate unpacking execution [40]. Themida [72], a well-known commercial packer, even applies virtualization obfuscation to its unpacking routine [85, 102], which will result in 630X instruction size explosion when tracing unpacking progress [63]. Worse still, malware authors can customize new packers from existing ones, as many packers are open source (e.g., UPX[2] and Yoda's Protector[3]). The lack of anti-analysis resistance renders existing generic unpacking futile for sophisticated packers. Security companies have been overwhelmed by packed malware over the past two decades, which slows down the response to emerging malware threats [15, 64, 73]. An online packing service even utilizes existing packers and anti-malware scanners as a feedback mechanism, and it returns the packer that presents the optimal evasion result [69].

In this paper, we present a new generic unpacking idea by studying how packers obfuscate payload binary's API call resolution. Our approach, named *BinUnpack*, is motivated by two key observations. The first one is, no matter how sophisticated a packer may evolve, malware payload always interacts with Windows OS to perform malicious behavior (e.g., code remote injection and ransomware's file encryption). Malware authors achieve this mainly by calling user-level Windows APIs rather than native APIs[4], since most API semantic information is missing at the native level. Typically, binary code resolves a Windows API's address by visiting PE header's IAT, which is an address lookup table when calling APIs exported by a dynamic-link library (DLL). As the Windows APIs in payload reveal rich semantics about malware and hence can provide security analysts with an upper hand, our second observation is a packer usually removes the payload's import address table (IAT) to impede reverse engineering. Afterwards, the unpacking routine will obtain each API's address and rebuild the IAT at run time. In this way, the restored payload can invoke Windows APIs properly.

These observations inspire us to chase down a new heuristic to determine the end of unpacking: if an API call is invoked through looking up a rebuilt IAT, it indicates that the original code has been restored, and the control flow has reached OEP already[5]. We call this property as *"rebuilt-then-called"*. The key idea of BinUnpack is to capture such a "rebuilt-then-called" feature instead of "written-then-executed" behavior. To this end, BinUnpack hooks API calls and finds the first one whose related IAT is rebuilt at run time. Then, tracing back from that API, BinUnpack is able to locate OEP within a very short distance. Compared to the existing work, BinUnpack presents a distinct advantage: it sidesteps multi-layer unpacking and avoids the significant overhead imposed by tedious memory access tracing. It seems API hooking has become a textbook problem, as many options are available in the arsenal [2, 101, 105]. However, state-of-the-art malware packers have already embedded anti-hooking tricks (e.g., stolen code [38], process hollowing [48], and crash hooking module [41]) to evade API monitoring. Our solution is to develop a hybrid, kernel-level DLL hijacking technique as API monitoring module. Our approach overcomes the path search order limit for core DLLs (e.g., kernel32.dll), which prevents the traditional way [79] from achieving complete DLL hijacking. Furthermore, we have integrated existing work in system call sequence alignment [42], memory subversion rootkit [90], and scalable binary

[1] Denis et al. prove that detecting the end of unpacking can be reduced to an NP-complete problem under certain assumptions [10].

[2] https://upx.github.io/

[3] https://sourceforge.net/projects/yodap/

[4] Windows system calls are also known as native APIs.

[5] We will discuss the exceptions of this conclusion in Section 7.

function matching [87] to deal with fake API calls, DLL integrity check, and custom API implementation, respectively. Our design enables a strong resistance to various evasions.

We have performed a large-scale evaluation with 238, 835 packed malware and a set of common Windows utilities. Our tested packer types contains a full range of malware packers in use, including sophisticated commercial and custom packers. We also evaluate BinUnpack with more challenging cases, such as multiple packer combinations, a partial code revealing packer[6], and possible denial-of-service (DoS) attacks. Nevertheless, BinUnpack maintains a high success rate consistently in all cases. The unpacked code produced by BinUnpack can greatly increase the accuracy of antivirus scanning, and the overhead to Windows utility execution is negligible as well. The comparative evaluation on a consumer grade laptop shows that BinUnpack outperforms existing tools in terms of significantly better performance and effectiveness. BinUnpack is able to complete unpacking within 0.5 second in most cases, which is substantially smaller than that of existing tools by *one* $\sim$ *three* orders of magnitude. The hook evasion evaluation indicates that BinUnpack outperforms well-known sandboxes (e.g., CwSandbox [101] and Cuckoo [71]) in terms of better resistance. Our encouraging results demonstrate that BinUnpack can be deployed to honeypot or sandbox to preprocess large-scale packed malware, or integrated into online malware scanning service such as VirusTotal[7] to achieve the optimal malware recognition rate.

Scope and Contributions: Another related obfuscation to binary packing is code virtualization [85, 102], which represents a completely different challenge. Although BinUnpack is immune to the case of unpacking routine virtualization (see Section 8.1.3), recovering virtualization protected code is out of our scope. In summary, the contributions of this paper are as follows.

- We propose a new, generic solution to quickly determine the end of unpacking by capturing the "rebuilt-then-called" behavior. Our approach is free from heavy memory access tracing caused by multi-layer unpacking, and therefore BinUnpack's performance is significantly better than the previous work.
- We design a novel, hybrid kernel-level DLL hijacking technique. We are not aware of any other scientific work on Windows core DLL hijacking. Our design enables BinUnpack to exhibit more powerful unpacking capability than the existing work.
- We evaluate BinUnpack extensively with large-scale datasets, which include almost all of the Windows malware packers available at present. BinUnpack shows consistently good results across various packers and potential attacks. A free online BinUnpack web service is under construction.

2 BACKGROUND AND MOTIVATION

In this section, we first summarize the drawbacks of existing work when dealing with multi-layer and anti-analysis packers. Then we present an example to illustrate another pervasive feature among packers: API call resolution obfuscation and import address table rebuilding. All of these inspire us to propose our unpacking method.

[6]It represents the worst case for all generic unpackers [7].
[7]https://www.virustotal.com

2.1 The Status Quo of Generic Unpacking

Existing generic unpacking approaches suffer from high overhead and lack of anti-analysis resistance. When a unpacking routine starts running, the procedure of writing to memory and then executing the written memory can repeat many times. We borrow the definition of unpacking layer from Xabier et al. [97]: "*a layer is, intuitively, a set of memory addresses that are **executed** after being **written** by code in another layer*". Their longitudinal study on 389 unique packers shows that 92.7% of them are multi-layer packers. Two factors contribute to the challenge of determining the end of unpacking. First, the "written-then-executed" feature is just an indication of dynamically generated code but not original code execution. Second, counter-intuitively, the original code is not necessarily in the deepest layer. CoDisasm [9] finds 19 out of total 28 packers have multiple layers. We further evaluate these 19 packers and identify that 4 of them do not present the original code at the last layer. Figure 1 illustrates such an example: the deepest layer only contains junk code rather than original code. Therefore, the unpacking heuristics that captures either the last layer or the signal of process termination (e.g., "TerminateProcess" or "ExitProcess") will miss the real payload. To follow the unpacking progress, the traditional approaches have to go through each unpacking layer via heavy memory access tracing and determine the presence of OEP with various heuristics. As a result, they typically impose significantly high runtime overhead and are too expensive for resource-constrained scenarios.

Generic unpacking utilizes various analysis systems to monitor unpacking progress. However, these analysis systems leak out many recognizable footprints, and packers can detect them to evade unpacking. For example, Armadillo will terminate execution in a debugging setting [21]; PESpin packers perform dynamic integrity check to fingerprint dynamic binary instrumentation environment [40]. Although some unpacking tools rely on hardware virtualization [19] to achieve transparency, the cost is a much higher performance penalty (e.g., 3, 000X slowdown [103]). The default function of Themida packer applies virtualization obfuscation to its unpacking routine, making monitoring unpacking progress even more difficult because of instruction size explosion [63]. Worse still, some anti-analysis tricks try to nullify the "written-then-executed" feature of memory pages [60] or attack the heuristics of original code identification [51]. In our comparative evaluation with other three recent unpacking tools, *no single previous work could cope with all of the tested packers.*

2.2 API Call Resolution

Binary packing technique keeps evolving itself to counter reverse engineering. But one thing maintains stable; that is, malware payload still needs to interact with Windows OS (via calling Windows APIs) to fulfill diversified malicious intents, such as process injection [30], C&C communication [26], and document encryption [43]. As compiler is unaware of DLL API addresses at compile time, a PE (Portable Executable) file has to resolve DLL API addresses dynamically, which comes in two major ways: 1) Type I: standard API resolution, a.k.a. implicit linking [59]; 2) Type II: dynamic API resolution, a.k.a. explicit linking [58]. Type I, the most prevalent way, accesses PE file header's import address table (IAT)

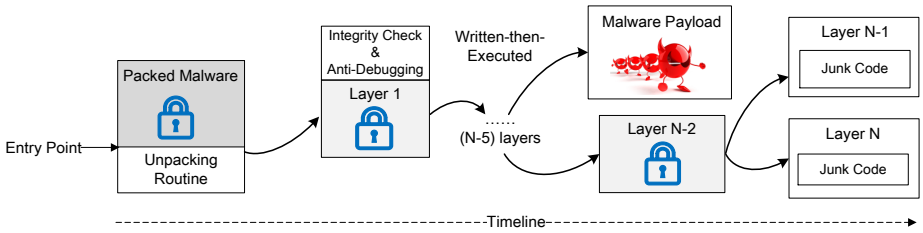


Figure 1: The unpacking process goes through multiple "written-then-executed" layers. The first layer contains anti-analysis code such as integrity check and anti-debugging, and the deepest layer does not consist of the malware payload but junk code.

to obtain an API address [84]. IAT entries list the function names or ordinals that need to be imported from a specific DLL. When a PE file is loaded, Windows loader is responsible for loading and linking required DLLs, and it also fills in the IAT entries with the virtual addresses of each imported function. The PE file refers to IAT by using indirect control flow instructions to call the DLL APIs. As a contrast, Type II has to make function calls to explicitly load the DLL and obtain an import function address at run time. The most convenient way is to explicitly invoke "LoadLibrary"[8] and "GetProcAddress"[9]. This means at least these two APIs are kept in IAT for Type II.

There are two exceptions to Type I & II in which IAT is not required. The first one is that API addresses can be hard-coded in binary. However, diversified Windows OS versions and address space layout randomization have put this exception into a dead end. The second exception happens at shellcode. As shellcode is not dynamically loaded by Windows loader but injected into the process space of victim program at run time, shellcode has to acquire the needed APIs' addresses without visiting IAT. Shellcode can first get the address of kernel32.dll from structured exception handling (SEH) or process environment block (PEB) structure [12], and it then searches the addresses of "LoadLibrary" and "GetProcAddress" from kernel32.dll's export directory. However, developing complicated malicious behavior using shellcode has many constraints and lacks compatibility [89]. That is the reason why shellcode is typically small and target-specific, and it is mainly used in the early stage of malware infection such as exploiting the vulnerability and bypassing the protection of data execution prevention. In addition, there are already mature solutions to prevent shellcode from obtaining the DLL address via SEH and PEB [92]. Therefore, we do not consider these two corner cases as practical.

2.3 Import Address Table Rebuilding

To complicate reverse engineering, packers obfuscate API call resolution by erasing the IAT of original code. Then the attached unpacking routine will rebuild a new IAT at run time, before resuming the original code execution. Rebuilding IAT means unpacking routine, like a custom dynamic loader, has to recover the connection from an API call name to its virtual address. This can be achieved by explicitly calling the API "LoadLibrary" and "GetProcAddress" (Type II). Note that a local IAT is attached to unpacking routine as well, as the unpacking routine itself has to call APIs for various purposes, such as detecting debugging/emulation environment and rebuilding the IAT of original code.

From code obfuscation viewpoint, removing IAT offers many benefits. First, many Windows APIs are abused for malicious purpose [20, 68, 88]. For example, "WriteProcessMemory" and "CreateRemoteThread" are often used together by malware authors to complete process injection [30]. Removing payload's IAT prevents a deep insight into the high-level semantics of malware. We manually modify an open source packer so that it does not remove the IAT of malware payload. The consequence is that another 14 additional anti-virus scanners are able to recognize this malware. The second benefit is to impede the reconstruction of a fully functional version of the original binary. In addition to removing IAT, many advanced packers go one step further to apply API redirection [39] during IAT rebuilding. For example, the address in IAT does not directly point to an API function but another memory region that has a direct jump to that API. API redirection hinders the perfect reconstruction of IAT, and therefore the unpacked code cannot function correctly. Another byproduct of deleting IAT is that it can further reduce the packed code size [82]. Several previous work has reported such IAT erasing and rebuilding behavior [17, 44, 82, 86, 97], but our work focuses on using these common features for generic unpacking.

2.4 Motivating Example

We use a malware sample hupigon.eyf[10] protected by FSG packer to illustrate the process of IAT rebuilding. Hupigon family was once notorious for the back doors they left on the compromised machine. Original hupigon.eyf contains 575 APIs, and it makes indirect calls to the API names stored in IAT (see Figure 2(a)). For the FSG packed version, FSG compresses code and data sections, erases the original IAT, and attaches an IAT to unpacking routine. As shown in Figure 2(b), the unpacking routine IAT contains only two API calls from kernel32.dll: "LoadLibrary" and "GetProcAddress", which are capable of rebuilding the IAT. Figure 2(c) is the memory view of FSG packed version at run time, and it also shows the common features that BinUnpack relies on. When the control flow arrives at OEP, the packed code and data sections have been restored, and the payload IAT, containing the same 575 APIs and their addresses, is reconstructed as well. Note that the reconstructed payload IAT is different from the unpacking routine IAT in two ways. 1) As the functionality of unpacking routine is relatively simple, it typically has much fewer imported APIs than the reconstructed payload IAT. 2) They reside in different memory regions. Recall that program

[8]It maps a DLL into a process's address space during execution.
[9]It returns an API call's virtual address.

[10]MD5:094578217633295012 73aa4659292401

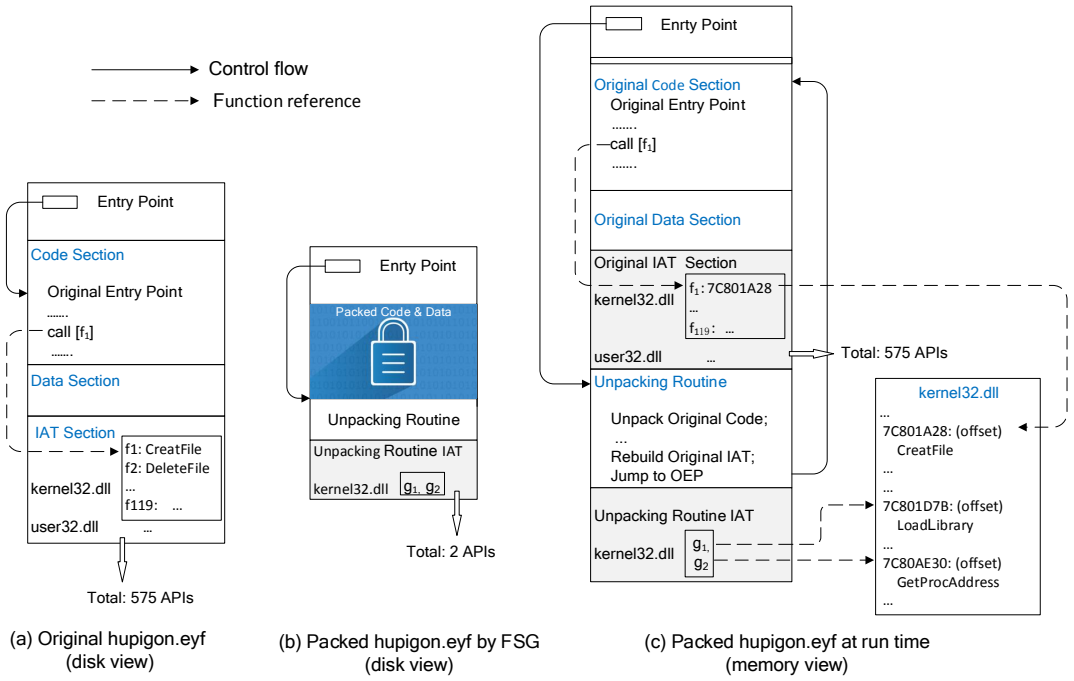


Figure 2: FSG packer removes the malware's original IAT from the packed version. The attached unpacking routine IAT only contains two APIs: "LoadLibrary" and "GetProcAddress", and they are enough to rebuild the original IAT.

refers to IAT via indirect calls (e.g., call $[f_1]$). Deliberately overlapping the payload IAT to the unpacking routine IAT's memory region is particularly challenging, because packers have to perform binary rewriting on the just recovered payload code and make sure all related indirect call addresses (e.g.,$f_1 \sim f_{119}$) are redirected to the new locations. BinUnpack takes advantage of these differences to detect the end of unpacking.

3 OVERVIEW

From the packers' pervasive IAT rebuilding behavior, we uncover two clues to help us determine the end of unpacking: 1) the reconstruction of payload IAT happens ahead of the jump to OEP; 2) at run time, if an API is *called* through a *rebuilt* IAT rather than the unpacking routine IAT, it indicates that malware payload has been restored. BinUnpack's key idea is to capture such *"rebuilt-then-called"* feature.

Figure 3 shows the architecture of BinUnpack. BinUnpack extracts the unpacking routine IAT of packed malware via static analysis, and then it monitors the dynamic execution of packed malware. The core of BinUnpack is "Hook-evasion Resistant API Monitor". It monitors API calls, find the related IAT to an API call, and compares the related IAT with the unpacking routine IAT ("Compare" in Figure 3). If the current related IAT is different from the unpacking routine IAT, that means the related IAT is rebuilt at run time. Next, BinUnpack halts the execution of packed malware and traces back to OEP ("OEP Search"). After that, we dump the memory of current process ("Process Dump") as BinUnpack's output ("Malware Payload"), which can be used for further malware analysis. Our design presents a distinct competitive advantage; that is, BinUnpack avoids the high runtime overhead caused by monitoring multiple "written-then-executed" layers. However, several other challenges are raised when we design BinUnpack's API monitor with existing methods [2, 101, 105]. All of them place BinUnpack in a dilemma: *they either can be easily evaded or are unaware of user-level API semantics*. Next section will discuss how we manage to address this dilemma in a hybrid way.

4 HOOK-EVASION RESISTANT API MONITOR

Another major contribution of BinUnpack is that our API monitor combines the existing two methods to achieve complete DLL hijacking. We rely on kernel-level hooking (method 1) to intercept an indispensable DLL loading function and load home-made DLL rather that target DLL (DLL hijacking, method 2). Our design amplifies the advantages of these two techniques and avoid their limitations. In addition, BinUnpack integrates several existing work [42, 87, 90] to defeat possible evasions and attacks.

4.1 API Hooking and Limitations

API hooking intercepts a call to an API function. The normal invocation flow will be rerouted to a different location where the hooking function resides. Existing API hooking methods [2, 101, 105] can be divided into two types: user-level and kernel-level hooking. User-level API hooking, such as IAT hooking and export address table (EAT) hooking, works at the user-level of OS and is process-specific. It has been adopted by many prevalent sandboxes [71, 101] to extract the user-level API semantics of malware. However, user-level hooking has to modify the target process space, which can be easily detected and countered by hook-evasion techniques [53]. Table 1

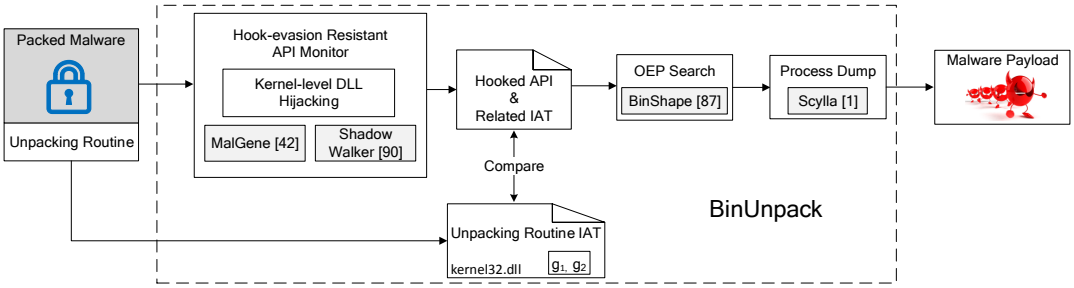


Figure 3: The Architecture of BinUnpack.

Table 1: Common user-level API hooking evasions adopted by packers.

Evasion Type	Packers
Stolen code	Asprotect, Pelock, Yoda's Protector, Yoda's Crypter, Enigma, Themida, Private exe Protector (PEP)
Child process	Armadillo, Pespin
Process hollowing	Ransomware custom packer
Crash hooking module	

presents the most common user-level API hooking evasions and representative packers we have encountered in our experiment. We divide them into three categories.

Stolen Code Stolen code [38, 82] copies some instructions from an API to an allocated memory in malware process. When a packed malware attempts to call the API, it first executes the copied instructions instead; and then the control flow jumps back to the API instruction which just follows the copied instructions. Stolen code happens at run time, after DLL has been loaded [82]. As many user-level API hooking techniques identify their target API calls by matching the virtual addresses where these APIs are expected to locate, stolen code will make such API hooking tools miss the target.

Child Process & Process Hollowing These two evasions are used to hide the presence of a malicious process. Child process means packed malware forks a child process, in which the malware executes unpacking routine and payload. User-level API monitoring is typically process-specific; that is, they only work at a specific process where the IAT or EAT is hooked. Some ransomware's custom packer has applied a more advanced technique, called "process hollowing" [48]. The effect of process hollowing is unpacking routine and original code execution will be decoupled into two processes. Any generic unpacking tool that does not have full control over multiple processes will be circumvented [44, 76].

Crash Hooking Module Some ransomware's custom packers have adopted a powerful anti-hooking technique: crash the hooking module [41]. The packer tries to create an access violation exception by arbitrarily calling APIs with invalid arguments. In a non-hooking environment, Windows OS default exception handlers can handle such errors, so packed ransomware can run properly. However, it is quite complicated to develop exception handlers for an API hooking system, and therefore it will crash when the access violation exception is raised. Security vendor VMRay [27] in May 2017 reported that the custom packer adopted by Cerber ransomware can crash all API hooking based sandbox solutions.

In contrast, kernel-level hooking of native API, is more difficult to be tampered with than user-level hooking, and it also has a global view over multiple processes. Unfortunately, kernel-level hooking does not suffice for BinUnpack, because there is no bijective mapping between user-level APIs and kernel-level native APIs [5]. Some user-level APIs such as path-related APIs and DLL management APIs (e.g., "GetProcAddress") provide user-level service exclusively. That means they do not invoke any native API at all. When BinUnpack is searching OEP (see Figure 6), we need to accurately hook "GetProcAddress" API so that we can limit the OEP search scope. Kernel-level hooking alone may miss the first API that is invoked from a rebuilt IAT or render the OEP search inaccurate.

4.2 DLL Hijacking

The limitations of user-level and kernel-level API hooking turn our attention to another API monitoring method: DLL hijacking [79]. As developers often load a common DLL by its name rather than its absolute path, DLL hijacking exploits the DLL path search order to load a custom-made DLL instead of the original DLL[11]. Compared to API hooking, DLL hijacking is more compatible with the target process. DLL hijacking withstands the evasion of stolen code. The reason is DLL hijacking does not modify the target process space, and the subject process has already loaded the custom-made DLL into its own space, before API instruction stealing occurs at run time. Therefore, the effect of stolen code is just like calling the custom-made API. Besides, DLL hijacking is immune to crash hooking module attack as it can naturally deliver runtime errors to Windows OS's exception handlers. However, given DLL hijacking's robust resilience, another problem rears its head.

Microsoft has realized that the default DLL path search order can be misused to load malicious component, so a more strict restriction comes up for core system DLLs such as kernel32.dll and advapi32.dll. The set of core DLLs and their full paths are explicitly specified by a particular Registry key [46]. We overcome the challenge of hijacking Windows core DLL by combining DLL hijacking and kernel-level hooking.

[11] Windows OS's standard DLL path search order and DLL hijacking example are shown in Appendix Table 5 and Appendix Figure 9.

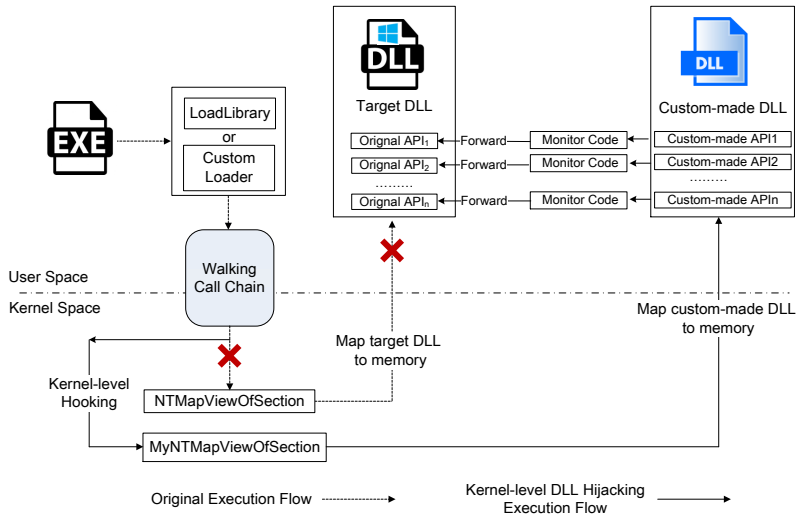


Figure 4: BinUnpack's kernel-level DLL hijacking.

```
1  LdrpMapDll()
2  {
3     //initialization code preparing for mapping DLL
4     initialization code here;
5    //map the DLL in the address of "BaseAddress"
6    NTSTATUS St= NtMapViewOfSection(...,*BaseAddress,...)
7     If(St==STATUS_IMAGE_NOT_AT_BASE)
8    {
9      //the DLL is not at base, relocate it.
10     relocation code here;
11     //initialization code preparing for mapping DLL
12     initialization code here;
13     /* map the dll in the address of "BaseAddress"
14      again, the "BaseAddress" may be modified by the
15      previous call of "NtMapViewOfSection"  */
16     NtMapViewOfSection(...,*BaseAddress,...)
17   }
18 }
```

Figure 5: The interaction between LdrpMapDll and NtMapViewOfSection. "BaseAddress" is the pointer to a base address where the DLL is mapped to.

4.3 Kernel-level DLL Hijacking

Standard DLL loading is mainly by means of calling "LoadLibrary" [46, 61], including implicit linking [59] and most cases of explicit linking [58]. This fact inspires us to bypass Windows path search order restriction. Particularly, we hook "LoadLibrary" and replace the core DLL with our custom-made DLL to achieve the goal of hijacking. As user-level API hooking does not resist to evasions, we switch to kernel-level hooking to intercept "LoadLibrary". We reverse-engineer Windows kernel with WinDbg [80] and find out a call chain from "LoadLibrary" to its related native API: "NtMapViewOfSection" in ntoskrnl.exe (the call chain is shown in Appendix Figure 10). The last user-level API before this call chain goes into kernel is "NtMapViewOfSection" function in ntdll.dll. "NtMapViewOfSection" in ntoskrnl.exe is the native API corresponding to "NtMapViewOfSection" in ntdll.dll. "NtMapViewOfSection" in ntdll.dll only forward all its parameters to "NtMapViewOfSection" in ntoskrnl.exe. Thus, we can use kernel-level hooking of "NtMapViewOfSection" in ntoskrnl.exe to intercept the "NtMapViewOfSection" in ntdll.dll. Figure 5 shows how "LdrpMapDll" interacts with "NtMapViewOfSection" in ntdll.dll. Note that "LdrpMapDll" calls "NtMapViewOfSection" twice at most. The first time happens at line 6. If the return value of this call is "STATUS_IMAGE_NOT_AT_BASE", it indicates the DLL has to be relocated to new memory space, and "NtMapViewOfSection" will be invoked again at line 16. The key of our method is to intercept the "NtMapViewOfSection" at line 6. And then, we redirect "*BaseAddress" to the memory loading address of our custom-made DLL. Also, we enforce "NtMapViewOfSection" returning "STATUS_IMAGE_NOT_AT_BASE" (line 7). In this way, the code from line 8 to 17 are activated, and our custom-made DLL will be loaded eventually. Appendix Algorithm 2 shows the detailed algorithm of loading custom-made DLL.

Figure 4 illustrates how we hijack standard DLL loading. We use the kernel-level hooking to intercept the original DLL loading flow and hijack the target DLL (including core DLL) with our custom-made DLL. "MyNtMapViewOfSection" intercepts the original control flow and maps custom-made DLL to memory. The monitor code in custom-made DLL conducts rebuilt IAT identification, OEP search, and process dump if necessary. If the IAT is not rebuilt at run time, the monitor code will forward the control flow to the real API in target DLL. Compared to other API hooking methods, our approach has many advantages such as being compatible with the target process and aware of user-level API semantics. These benefits enable BinUnpack to reveal better resilience to the common hook evasions adopted by packers (see Table 1).

4.4 Non-Standard Explicit Linking

We have to consider the non-standard implementations of explicit linking, because malware authors have already adopted them to evade the hooking of "LoadLibrary". The first way is to re-implement the functionality of "LoadLibrary" by calling "CreateFileMapping" and "MapViewOfFile" [57]. However, this custom loader still eventually invokes "NtMapViewOfSection" to load DLL. Our kernel-level hooking of "NtMapViewOfSection" is capable of

dealing with this case. Another recent work, Stealth Loader [39], avoids the use of file-map APIs such as "CreateFileMapping" via reflective DLL injection technique [22]. It calls "CreateFile", "ReadFile", and "VirtualAlloc" to map a DLL into non-file-mapped memory at the expense of large memory footprint. This means Stealth Loader does not go through "NtMapViewOfSection" at all. However, Stealth Loader's trace is not invisible. For example, it calls "CreateFile ("kernel32.dll")" to open a DLL before it allocates virtual memory for the DLL. We captures this characteristic feature by kernel-level hooking of "NtCreateFile" to open our home-made DLL. In this way, we can monitor the APIs loaded by Stealth Loader.

5 IAT COMPARISON

As shown in Figure 4, the custom-made DLL in BinUnpack is used to mimic the target DLL we want to hijack. We automatically generate the custom-made DLL from the target one and attach the monitor code to each custom API. Algorithm 1 uses the "DeleteFile" in kernel32.dll as an example to show how the custom-made DLL works. We follow the similar approach with QuietRIATT [78] to get current IAT using hooked DLL calls (line 6). The key idea is to find indirect call pattern via stack backtrace [24]. Line 7 compares two IATs in terms of different memory locations or contents.

One complicated case to IAT comparison is that an attacker could apply multiple packer combinations. That is, the innermost packed code and unpacking routine are further packed by another packer. Similarly, the IAT of inner unpacking routine has to be first erased and rebuilt later. In this case, we will see the behaviors of multiple phases "rebuilt-then-called". Therefore, we add a new global variable "lastIAT" in Algorithm 1 to represent the last rebuilt IAT. The initial value of "lastIAT" is the unpacking routine IAT of the outermost packer (line 2–4). If "currentIAT" is different from "lastIAT", it indicates there is a new phase of "rebuilt-then-called". Otherwise we forward the execution flow to the original API to continue the execution of unpacking routine (line 14). Following line 7, we perform backtrack search for OEP (line 8). If we find OEP (line 9), we will first restore the unpacked code protected by this packer (line 10) and then update "lastIAT" (line 11) for the next round comparison. Following this style, Algorithm 1 is able to recover the packed code protected by each packer and finally get the original malware payload. In our evaluation, handling multiple packer combinations only leaves little impact on BinUnpack's performance, but it imposes significant performance degradations on traditional generic unpacking tools (see Table 3).

6 OEP SEARCH & PROCESS DUMP

The original entry point (OEP) is the first instruction of the restored code. A wrong OEP will mislead a disassembler to produce incorrect instruction. Although the previous work has proposed many effective heuristics to find the existence of OEP such as standard compiler signature [36], dangerous API call [28], cross-section jump [76], and yara rules [55], they still suffer from very large OEP search space. Recall that the reconstruction of payload IAT completes before the recovered payload resumes execution. We can utilize the API calls respectively from unpacking routine and payload to narrow down the OEP search space. As shown in Figure 6, we

Algorithm 1 Custom-made API (MyDeleteFile)

lpFileName: The name of the file to be deleted.
lastIAT: A global variable representing the last rebuilt IAT.

```
1:  function MyDeleteFile(lpFileName)
2:      if lastIAT = ∅ then
3:          routineIAT ← GetUnpackingRoutineIAT()
4:          lastIAT ← routineIAT
5:      end if
6:      currentIAT ← GetCurrentIAT()
7:      if currentIAT ≠ lastIAT then // IAT Comparison
8:          OEP ← BacktrackOEP()
9:          if OEP ≠ ∅ then
10:             ProcessDump()
11:             lastIAT ← currentIAT
12:         end if
13:     end if
14:     return DeleteFile(lpFileName)
15: end function
```

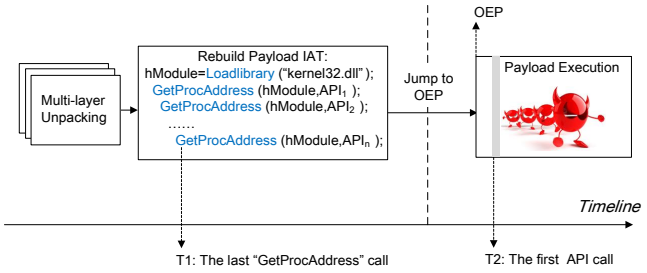


Figure 6: API monitoring limits the search scope of OEP. Kernel-level hooking alone will miss the checkpoint "T1".

first find the checkpoint for the last "GetProcAddress" used to rebuild the payload's IAT ("T1" in Figure 6), and then we identify the checkpoint for the first API call from the payload ("T2"). Obviously, the memory search scope of OEP is limited by the upper bound "T1" and the lower bound "T2". Next BinUnpack backtracks from "T2" to 'T1" to search OEP using the same heuristics as Arancino unpacker [76]. Our approach can reduce the possible OEP search scope remarkably. For example, when unpacking the hupigon.eyf protected by Armadillo, PinDemonium [55] has to search as much as 21, 548 instructions before locating OEP. By contrast, BinUnpack only goes through 19 instructions. In our large-scale evaluation, the maximum number of instructions from "T2" to OEP is 168, and the average value is 32.

Home-Made API Implementation Like the custom loader we have discussed in Section 4.4, malware authors can also implement the functionality of "GetProcAddress" by searching DLL module's export directory [57]. In this way, hooking "GetProcAddress" will miss the checkpoint "T1". As "GetProcAddress" does not invoke any native API, kernel-level hooking does not work either. We adopt existing work in scalable and obfuscation-resilient binary library function matching, BinShape [87], to fast identify the specific memory patters of API custom implementation. BinShape derives heterogeneous features to represent library function, covering control flow graph, instruction-level, and statistical features. The detailed evaluation data is shown in Table 7.

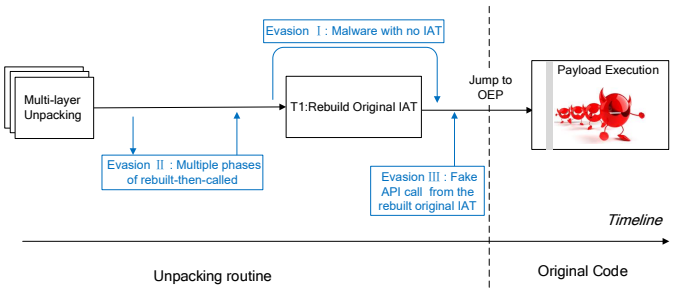


Figure 7: The possible evasions against "rebuilt-then-called" behavior.

Many packers also adopt anti-dumping tricks to prevent obtaining unpacked code from memory. One common way is to modify the access attribute of PE head in memory to "NO_ACCESS". As a result, the dumping tools will crash when access the PE head. BinUnpack relies on the state-of-the-art process dump tool, Scylla [1]. In our evaluation, Scylla achieves the optimal results when handling anti-dumping techniques. With the original payload produced by process dump, applying further malware code analysis such as binary code disassembly [9, 45], binary diffing [23, 63], and large-scale malware clustering/lineage [29, 32] becomes straightforward.

7 POSSIBLE ATTACKS AND COUNTERMEASURES

Although BinUnpack is conceptually simple, we show in Section 8 that it is able to recover the original code with very high accuracy and efficiency. Even so, we have to consider how a skilled attacker could circumvent BinUnpack once our approach is known. This section discusses possible attacks and our countermeasures.

7.1 Attacks to "Rebuilt-then-Called"

The key idea of our approach is to quickly determine the end of unpacking by capturing "rebuilt-then-called" behavior. Figure 7 shows three possible ways to evade/attack this key feature.

Malware with no IAT (Evasion I in Figure 7) In Section 2.2, we have discussed two exceptions that malware can have no IAT and their practical constraints. Developing malware samples with hard-coded API addresses or shellcode can hinder their executions and propagations on diverse victim machines. Therefore, we consider these two corner cases too unreliable to be a real threat.

Fake Rebuilt IAT and Fake API Call Like the multiple "written-then-executed" layers in a single packer, an intuitive attack to BinUnpack is also generating multiple phases of "rebuilt-then-called". For example, a single packer can rebuilt a fake IAT and call APIs from it (Evasion II in Figure 7). Similarly, after rebuilding the original IAT, the packer can invoke a fake API call before control flow jumps back to OEP (Evasion III in Figure 7). Note that both Evasion II and Evasion III can only trigger BinUnpack searching OEP but no process dump, because no OEP will be detected in these two evasions (see Algorithm 1). As BinUnpack's OEP search is efficient (see Section 6), limited times of Evasion II and III do not have much of an impact on BinUnpack's performance. However, a determined attacker can perform a Denial-of-Service (DoS) attack; that is, performing many iterations of Evasion II & Evasion III.

We do not treat this extreme case as a hard limit. Compared to multiple "written-then-executed" layers, multiple phases of "rebuilt-then-called" will impose dramatically large overhead to packed malware itself, because API calls are much more expensive than the read and write instructions. To evaluate the impact of DoS attacks to BinUnpack, we have simulated Evasion II & Evasion III in the open-source UPX packer. As shown in Table 2, the BinUnpack's overall running time is pretty small when the iterations of Evasion II & III are less than 10^6. When the iteration number reaches 10^9 times, the DoS-UPX will introduce as much as 10^6 X additional slowdown, and BinUnpack will be occupied by OEP search. To proactively mitigate this possible attack, we have applied an advanced bioinformatics-inspired system call sequence alignment, MalGene [42] to bypassing fake API calls in two runs. Particulary, in the first run, we only enable BinUnpack's API monitor function and MalGene so that MalGene can identify the scope of Evasion II and Evasion III. In the second run, we only enable BinUnpack's unpacking function when the current API is not in the scope of Evasion II or Evasion III. In this way, for the extreme case of 10^9 iterations, we can reduce BinUnpack's running time to less than 158 minutes.

7.2 Attacks to Kernel-level DLL Hijacking

As the core of BinUnpack's API monitor is kernel-level dll hijacking, this section discusses possible attacks to this component.

Kernel-level Hooking Detection It has become much harder for malware to be loaded into the kernel space and defeat BinUnpack's API monitoring. Since Windows 64-bit Vista released in 2007, Microsoft has employed a new security mechanism, called "Mandatory Driver Signing", to prevent the OS kernel hacked by malware [75]. "Mandatory Driver Signing" requires that all kernel-mode drivers to be digitally signed to verify code integrity.

DLL Integrity Check Another possible evasion is to check the integrity of a DLL. As Microsoft's DLLs such as kernel32.dll are publicly available, the packer could calculate kernel32.dll's hash value offline and compare with the hash value of the dll in memory at run time. Since BinUnpack has substituted the kernel32.dll at load-time with our custom-made DLL, whose hash value is different from the original one. However, the released Windows OS has too many different versions, and verifying all possible hashes is not a trivial task for attackers either. Nevertheless, we have extended BinUnpack with a memory subversion technique [25], Shadow Walker rootkit [90], to bypass DLL integrity checking. The basic idea is to forward the data access of the custom-made DLL to the target DLL and the code access of the custom-made DLL to itself. Therefore, the hash value used in integrity check is calculated from the target DLL instead of the custom-made DLL.

8 EVALUATION

We collect total 271,095 malware samples from three different malware repositories: VX Heaven[12], VirusShare[13], and VirusTotal. These malware samples cover major malware categories such as

[12] http://vxheaven.org/
[13] http://virusshare.com/

Table 2: Running time (seconds) of Evasion II & Evasion III DoS attacks. They have relatively small impact on BinUnpack.

Sample	No BinUnpack (s)	BinUnpack (s)				Relative Slowdown
		IATComparison	OEPSearch	Dump	Total	
UPX	0.12	2.1×10^{-5}	3.5×10^{-6}	3.1×10^{-2}	3.1×10^{-2}	25.8%
UPX+(Evasion II & III)$\times10^3$	0.24	2.3×10^{-3}	7.4×10^{-3}	3.1×10^{-2}	4.1×10^{-2}	17.1%
UPX+(Evasion II & III)$\times10^6$	122	2.2	7.3	3.1×10^{-2}	9.5	7.9%
UPX+(Evasion II & III)$\times10^9$	122,472	2,231	7,264	3.1×10^{-2}	9,495	7.8%

backdoor, worm, trojan, and virus, including now-infamous ransomware families. The active time of these samples ranges from 2008 to 2018. As far as we know, our work is the first one to evaluate generic binary unpacking on such a scale. Our testbed is a consumer grade laptop with an Intel Core i3-36100 processor (Quad Core, 3.70GHz) and 8GB memory, running Windows 7. We have used packer signature matching tool Exeinfo PE[14] and PEiD[15] to rule out the non-packed samples and the ones that are only protected by code virtualization, which is out of BinUnpack's coverage. Eventually, we obtained 238, 835 packed malware binaries. 75.8% of them are from known packers, and the left (24.2%) are protected by unknown packers. We also find that custom packers have been quite common among new-generation malware such as ransomware.

8.1 Comparative Evaluation

We perform comparative evaluation to accurately evaluate BinUnpack's effectiveness and performance with the malware samples that we have ground truth. That is, we either have their source code or we are able to manually verify the unpacking result. Table 3 shows the results on a set of packed versions of our motivating example—hupigon.eyf. This experiment represents a typical scenario that malware authors generate new variants by applying different packers. They may select the optimal one that achieves the best evasion effect to propagate.

The first column of Table 3 lists all of the packers we have tested, including known packers and packer combinations. The second column shows "the number of layers/original code layer" for these packers. We can see that most of the packers are multi-layer packers, and some packers do not reveal the original code in the deepest layer (the number in bold). For example, ACProtect packer has the most 216 layers, but the original code locates at the second layer. The numbers in the third column represent API numbers in the unpacking routine IAT, which confirms that erasing the payload IAT is very common for packers. All of these numbers are less than the number of hupigon.eyf's imported APIs (575), and many packer IATs have only two APIs: "LoadLibrary" and "GetProcAddress". The fourth column lists the common evasion types applied by these packers. Hupigon.eyf belongs to a large family of backdoor Trojan, and their main function is to form a botnet by connecting a number of victimized machines. Since the first detection of Hupigon goes back to 9 years ago[16], we believe most anti-malware scanners are able to recognize it. We send the hupigon.eyf's no-packer version to VirusTotal, and there are total 36 anti-virus products correctly label our submission as "hupigon". Therefore, we treat 36 as the optimal anti-malware scanning result.

[14]http://exeinfo.atwebpages.com/
[15]https://www.aldeid.com/wiki/PEiD
[16]https://en.wikipedia.org/wiki/Backdoor:Win32.Hupigon

8.1.1 Multi-layer Packers. These packers are all single packer but with multiple "written-then-executed" layers. Column 5 shows that these packers can circumvent many anti-virus products. Although some advanced anti-virus engines have already embedded a sandbox for generic unpacking [34], there is *no single* one that could cope with all of the packed samples in Table 3. Column 6 presents the VirusTotal results for the four unpacking tool outputs, and only BinUnpack's numbers are very close to the optimal value (36). Compared to other tools, BinUnpack significantly improves the accuracy of anti-virus scanning. For the packers that do not reveal the original code in the deepest layer (e.g., Enigma, SoftwarePassport, Armadillo, and ACProtect), some VirusTotal results of the unpacked code are 0. The reason is these unpackers do not return the right layer containing the original code.

8.1.2 Packer Combinations. With different packers applied on the same original code repeatedly, generic unpacking becomes more difficult. On the other side, the packed code generated by some packers may not be further packed by other ones [28]. After trying all possible combinations, we find several successful cases. Compared to a single packer, packer combinations are indeed more likely to evade anti-malware scanning. VirusTotal's detection number for these packer combinations is no more than 14. As our design has already considered the threat of packer combinations (see Algorithm 1), BinUnpack can efficiently extract original code from multi-packer protected version.

8.1.3 Themida. Themida is a sophisticated commercial packer, and it is also widely used by malware authors [85, 102]. As shown in Column 5, the detection rate to Themida protected malware is rather low. We use Themida to evaluate two more complicated cases. First, we apply Themida's default binary packing function to hupigon.eyf's binary code. The distinct feature of Themida packer is that the unpacking routine code is further protected by virtualization obfuscation. In this case, one unpacking routine instruction is replaced by several bytecode, and the attached virtualization engine will simulate these bytecode at run time. Under unpacking routine virtualization, tracing "written-then-executed" instructions will become extremely time-consuming [63] . Unlike the previous work, BinUnpack neatly sidesteps heavy memory access tracing and avoids the high overhead caused by code virtualization execution.

Second, as the source code of hupigon is available, we can enable Themida's optional function "Encode Macro". "Encode Macro" allow users to mark a region of source code to be encrypted. When the encrypted code region is to be executed, Themida will first decrypt the code inside the macro, execute it, and then encrypt it again. Themida with "Encode Macro" option enabled can be taken as a partial code revealing packer, which represents the worst case for all generic unpackers [7, 86, 97] because only a portion of original

Table 3: Comparative evaluation with ground truth dataset. Evasion types: 1) Anti-Debugging; 2) Anti-VM (Virtual Machine & System Emulator); 3) Anti-DBI (Dynamic Binary Instrumentation); 4) Anti-Hooking. The last two columns's order is (CoDisasm, PINdemonium, Arancino, BinUnpack).

Packers	#Layers	#APIs	Evasions	VirusTotal Result		Performance (s)
				Packed	Unpacked	(CoDisasm, PinDemonium, Arancino, BinUnpack)
hupigon.eyf (no packer)		575			36	
Multi-layer packers						
NsPack	2/2	28		19	0, 8, 8, 35	T, 312, 343, 0.15
nPack	2/2	5		22	0, 26, 26, 36	E, 17, 21, 0.09
FSG	2/2	2		24	0, 21, 21, 36	T, 27, 42, 0.10
UPX	2/2	5		30	21, 26, 26, 36	3, 16, 23, 0.13
eXPressor	2/2	12		22	0, 24, 24, 35	T, 9, 15, 0.11
RLPack	2/2	6		21	0, 6, 6, 35	T, 10, 25, 0.12
Petite	3/3	30		21	27, 29, 29, 36	2, 71, 100, 0.11
Aspack	3/3	6		31	0, 21, 21, 36	T, 80, 105, 0.09
MoleBox	3/3	144	1	22	0, 32, 32, 36	T, 64, 82, 0.17
Asprotect	3/3	6	1,4	20	0, 19, 19, 34	T, 162, 180, 0.21
WinUpack	3/3	2		22	10, 19, 19, 33	9, 13, 25, 0.09
FishPacker	3/3	2		23	0, 21, 21, 32	T, 69, 82, 0.10
KBys	3/3	4		23	0, 8, 8, 31	T, 83, 105, 0.15
PECompact	4/4	4		24	0, 12, 12, 36	T, 13, 20, 0.11
Yoda's Crypter	4/4	2	1,4	19	14, 0, 0, 35	7, E, T, 0.16
MEW	4/4	2		24	0, 21, 21, 34	T, 72, 105, 0.14
ORiEN	4/4	4		25	23, 26, 26, 35	6, 56, 64, 0.13
PEP	4/4	310	1,4	12	0, 0, 0, 28	T, T, T, 0.20
Enigma	**5/4**	18	1,2,4	11	0, 0, 0, 34	T, E, T, 0.17
ZProtect	5/5	20		18	0, 9, 9, 31	T, 34, 56, 0.17
Yoda's Protector	6/6	2	1,3,4	21	19, 0, 0, 35	8, E, T, 0.11
Obsidium	6/6	2	1,3	13	0, 6, 6, 31	T, 168, 205, 0.15
SoftwarePassport	**7/6**	242	1,2	21	0, 0, 0, 33	T, T, T, 0.36
Pelock	15/15	22	1,4	10	0, 0, 11, 35	T, E, 256, 0.32
Telock	18/18	2	1,3	21	15, 0, 5, 32	13, E, 105, 0.18
Pespin	80/80	4	1,3	16	0, 0, 18, 32	E, E, 436, 0.14
Armadillo	**165/163**	232	1,2,4	13	0, 0, 0, 28	T, T, T, 0.38
ACProtect	**216/2**	5	1,3	9	0, 0, 0, 31	E, E, 256, 0.29
Packer combinations						
NsPack+Aspack	5/5	28		10	0, 7, 7, 34	T, 380, 421, 0.17
RLPack+MoleBox	5/5	6	1	8	0, 5, 5, 33	T, 71, 101, 0.25
nPack+PECompact	6/6	5		9	0, 8, 8, 34	E, 24, 38, 0.15
eXPressor+MoleBox	5/5	12	1	13	0, 23, 23, 33	T, 68, 91, 0.23
RLPack+Aspack	5/5	6		11	0, 5, 5, 34	T, 83, 126, 0.15
FishPacker+PECompact	7/7	2		13	0, 7, 7, 30	T, 79, 94, 0.15
FishPacker+Aspack	6/6	2		14	0, 18, 18, 30	T, 141, 181, 0.13
FishPacker+PEP	7/7	2	1,4	11	0, 0, 0, 26	T, T, T, 0.25
Themida						
Unpacking routine virtualization	**106/105**	2	1,2,4	8	0, 0, 0, 31	T, E, E, 0.75
Partial code revealing	**107/105**	2	1,2,4	6	0, 0, 0, 30	T, E, E, 0.85

[1] "T"(Timeout) means that the unpacking tool running time exceeds 1,800 seconds or 120 seconds for CoDisasm.
[2] "E"(Exception) means that the unpacking tools raise exceptions and then exit.

code is revealed during any given unpacking time window. Our countermeasure is to dump a continuous series of process memory and later reassemble them as a single consistent code image. We enable Themida's "Encode Macro" option to protect major functions of hupigon.eyf. This means we can only restore one function's binary code each time. As shown in the last row, BinUnpack is able to extract the original code successfully and greatly increase malware detection rate. However, in practice, such partial code revealing packer is rare because of no source code available, the unreliability, and high runtime overhead [7].

8.1.4 Performance Comparison. We also compare BinUnpack's performance with other three representative generic unpacking tools: CoDisasm [9], PinDemonium [55], and Arancino [76]. All of them rely on Pin [52] to monitor "written-then-executed" instructions [55, 76] or memory pages [9]. We borrow "CoDisasm", "PinDemonium", and "Arancino" to represent the generic unpackers developed by the related work [9, 55, 76], respectively. Although the purposes of these three papers look different (e.g., binary disassembly [9] or defeating anti-instrumentation evasions [76]), generic unpacking is either an indispensable preprocess or an appealing application. BinUnpack's two major advantages are avoiding tracing "written-then-executed" layers and resilient to anti-analysis tricks. All of these three unpackers are very representative in these two respects. Like BinUnpack, PinDemonium also relies on the state-of-the-art process dump tool, Scylla [1], to reconstruct the payload IAT. Arancino is an extension of PinDemonium[17], and it is the latest work that is able to defeat anti-DBI-equipped packers.

In our evaluation, we set the threshold of runtime execution as 1,800 seconds[18]. When an unpacking tool exceeds this threshold, we will stop the execution and mark the performance data as "Timeout". Besides, we also find many cases that the unpacking tools raise exceptions and then exit, and therefore we mark them as "Exception". The last column of Table 3 shows that BinUnpack succeeds in all cases, and the running time ranges from 0.09 to 0.85 seconds with an average value 0.20. The worst case comes from unpacking partial code revealing packer. In contrast, the overhead of other three tools is much larger than BinUnpack by *one* $\sim$ *three* orders of magnitude. In addition, all of them fail in many cases with either "Timeout" or "Exception". Especially, they fail in most packer combination samples and Themida versions. We attribute their failures to the lack of anti-analysis resistance. For example, PESpin and ACProtect packers can fingerprint Pin environment [40] and crash the execution of CoDisasm and PinDemonium. Arancino [76] is an extension of PinDemonium [55] to defeat anti-instrumentation-equipped packers at the cost of much more overhead.

[17] They are developed by the same team.
[18] CoDisasm provides a server to generate traces and memory dumps. The server's "Timeout" threshold is only 120 seconds.

For some other representative generic unpacking tools, as many of them are either unavailable or obsolete, we report the overhead mentioned in their papers [28, 36, 56, 83, 86]. Note that all of them only evaluated limited packers and no one tested packer combinations or the partial code revealing packer. Whether they can still defeat current sophisticated packers is unclear. The overhead of PolyUnpack [83] is about 150 seconds on average; The average unpacking time of OmniUnpack [56] on 12 packers is 5.3 seconds; Renovo [36] incurs at least 8X runtime slowdown when unpacking 15 packers; Eureka [86] tested 15 packers and could unpack "more than 90 binaries per hour".

8.1.5 Top Wanted Malware, Packed Benign Programs, and NSIS Packer. In addition to hupigon.eyf, we also evaluate top wanted malware, packed web browsers, and custom packers adopted by ransomware. The top wanted malware samples are from the security vendor Check Point's top10 list from May 2017 [95] to March 2018 [96], and we manage to collect their binary code of no-packer versions. The second column of Table 4 shows the number of various packers applied to each sample. We find that not all of the packers list in the Table 3 are compatible with top wanted malware and benign browsers. The Column 3∼ 5 list the average VirusTotal scanning results to no-packer, packed, and unpacked versions, respectively. Column 6 shows the number of successful unpacking. We consider an unpacking as success if it can locate OEP and extract the original code correctly. The last column reports the average running time for successfully unpacking cases. For the experiments of packed top wanted malware, BinUnpack's results are similar to Table 3 with *two* ∼ *three* orders of magnitude performance boost and 100% success rate.

In the packed benign program experiments with three popular web browsers, we find that more that 20 anti-virus scanners generate false alarms, and all of the other three unpacking tools fail to extract the original code. BinUnpack's outputs reduce the false positives of anti-virus scanners to *zero*. In the last set of experiments with custom packers adopted by ransomware, the third column data is not available because we do not have their no-packer versions. We find all of these ransomware samples customize NSIS installer as packers. NSIS (Nullsoft Scriptable Install System) is a script-driven installer to create a install package. As a lot of legitimate softwares also use NSIS as their installers, anti-virus scanners would produce large false positives if they take NSIS as malware signature. The unique feature of the NSIS packer protected malware is that it never places the malicious executables on the file system (a.k.a Fileless Malware [106]), which can bypass many malware static analysis approaches. Appendix Figure 11 shows the typical attacking procedure of NSIS packed ransomware. To further complicate the generic unpacking, Cerber customizes NSIS packers by adding evasion techniques such as process hollowing and crash hooking module. Nonetheless, we find that the NSIS packed ransomware still presents the behavior of "rebuilt-then-called", so BinUnpack is able to extract the original payload efficiently.

8.2 Impact on Benign Program Execution

We also study the overhead BinUnpack introduces when working with benign programs (no-packer version). We compare the execution time of benign programs with BinUnpack disabled and enabled.

Table 4: Comparative evaluation summary with more samples. The data in the last three columns is represented as (CoDisasm, PinDemonium, Arancino, BinUnpack).

Sample	#Packers	VirusTotal (Avg.)			#Success	Peformance (Avg.)
		No-packer	Packed	Unpacked		
Top wanted						
Locky	32	38	15	3, 11, 13, 35	5, 18, 21, 32	101, 488, 512, 0.21
Conficker	35	47	25	3, 10, 13, 45	6, 19, 23, 35	105, 532, 568, 0.28
Zeus	36	40	18	3, 11, 15, 37	7, 21, 24, 36	92, 410, 451, 0.24
Andromeda	33	35	11	3, 11, 14, 31	6, 18, 22, 33	90, 380, 446, 0.21
Necurs	31	37	13	3, 10, 13, 33	6, 18, 21, 31	76, 364, 395, 0.17
GlobeImposter	34	48	24	3, 11, 14, 45	7, 19, 23, 34	93, 415, 450, 0.20
Pykspa	35	42	19	2, 10, 12, 37	5, 18, 21, 35	93, 423, 467, 0.21
Hancitor	34	46	22	3, 12, 15, 42	7, 20, 24, 34	79, 358, 394, 0.18
Nivdort	37	30	11	3, 11, 14, 26	8, 22, 25, 37	92, 405, 462, 0.20
WannaCry	32	56	30	4, 10, 13, 54	6, 17, 20, 32	120, 523, 576, 0.26
Kelihos	38	32	12	3, 11, 14, 28	7, 22, 26, 38	91, 401, 452, 0.22
Jaff	33	43	19	3, 10, 12, 40	6, 18, 21, 33	92, 422, 473, 0.20
Cryptowall	31	44	22	3, 10, 12, 41	5, 17, 19, 31	114, 434, 482, 0.25
Sality	35	43	19	3, 11, 13, 40	7, 20, 24, 35	95, 461, 514, 0.21
Fareit	36	46	17	4, 13, 16, 43	9, 23, 27, 36	94, 412, 452, 0.20
Packed browsers						
IE	9	0	23	–, –, –, 0	0, 0, 0, 9	E, T, T, 0.29
FireFox	5	0	20	–, –, –, 0	0, 0, 0, 5	E, T, T, 0.31
Chrome	8	0	22	–, –, –, 0	0, 0, 0, 8	E, E, T, 0.33
Custom packers						
CryptoLocker	1	–	34	–, –, –, 44	0, 0, 0, 1	T, E, E, 0.23
CTB-Locker	1	–	31	–, –, –, 41	0, 0, 0, 1	T, E, E, 0.22
Teerac	1	–	21	–, –, –, 32	0, 0, 0, 1	T, T, T, 0.19
Crysis	1	–	22	–, –, –, 29	0, 0, 0, 1	T, T, T, 0.21
Cerber	1	–	36	–, –, –, 44	0, 0, 0, 1	E, E, E, 0.25

[1] The meaning of "T"(Timeout) and "E"(Exception) are as the same as Table 3, and "–" means N/A.

The test set includes common Windows applications (e.g., tasklist, winrar, and WinPcap) and prevalent web browsers (e.g., IE, Firefox, and Chrome). To evaluate the browsers, we cache the top 10 benchmark sites ranked by Alexa[19]. We insert the JavaScript "getTime()" routine to the start and the end of the page, and then compute the delta between these two time. The delta shows us how long it takes to load a page. As shown in Appendix Table 6, the additional overhead caused by BinUnpack mainly comes from the IAT comparison. If the result of IAT comparison is true, BinUnpack will not perform both OEP search and process dump. The relative slowdown caused by BinUnpack ranges from 0.01% to 1.48%. The worst case comes from Chrome browser, which heavily uses API calls (e.g.,"WriteFile"). So BinUnpack has to perform IAT comparison frequently. Overall, BinUnpack only brings marginal overhead to benign program execution.

8.3 Hook-evasion Resistance Evaluation

In addition to high runtime overhead, the lack of anti-analysis resistance is another reason to limit the application of generic unpacking. In this section, we evaluate the capacity of BinUnpack's hook-evasion resistance with a classical user-level API monitor (Detours [33]) and two prevalent sandboxes that provide API hooking service (CWSandbox [101] and Cuckoo Sandbox [71]). The test data contains the known packers from Table 1, which lists common hook evasions adopted by packers. Besides, we modify the source code of UPX to represent another three evasions: Custom Loader [57], Stealth Loader [39], and DLL integrity check. Custom Loader [57] implements the functionality of "LoadLibrary" and "GetProcAddress", which avoids explicitly calling these two APIs. Stealth Loader [39] avoids calling "NtMapViewOfSection" and maps a DLL into non-file-mapped memory. DLL integrity check is used

[19] http://www.alexa.com

to detect whether the dll loaded in memory is modified. Fortunately, BinUnpack has countermeasures to defeat all of these evasions.

The evaluation results are summarized in Appendix Table 7. For child process and process hollowing evasions, CWSandbox and Cuckoo can deal with them because these sandboxes recursively hook APIs in child processes. But Detours is process-specific, and it needs to inject code into the target process space. Except BinUnpack, the left API monitors identify target APIs by matching the virtual addresses where these APIs are expected to locate. Therefore, stolen code technique can make them miss the target. As Detours and the two sandboxes do not implement exception handlers perfectly, crash hooking module is particularly effective to impede all of them. In addition, both Custom Loader [57] and Stealth Loader [39] are able to circumvent user-level API hooking. By contrast, the design of kernel-level DLL hijacking and fast binary function matching technique [87] enable BinUnpack to resist to these stealthy hook evasions. Furthermore, only BinUnpack is resilient to DLL integrity check. Other three API monitors also have to patch the DLL loaded in memory. Section 7.2 has discussed the countermeasure we have taken to bypass DLL integrity checking; that is, we adopt Shadow Walker rootkit [90] to forward the data access of custom DLL to the target DLL. Strong hook-evasion resistance explains why BinUnpack exhibits much broader unpacking scope, and our approach also provides a new way to develop resilient API monitor in sandbox.

8.4 Unpacking Wild Packed Malware

The high performance of BinUnpack enables us to perform a large-scale evaluation. We test BinUnpack on 238, 835 packed malware samples we have collected. BinUnpack succeeds in 97.3% of samples, and each unpacking can be completed within 0.5 second for most cases. The reason for the left 2.7% failures is either the binary code is not executable or the unpacking routine exits early. After further investigation, we find some custom packers attempt to detect the involvement of human by checking the movement of mouse cursor. If the positions of mouse cursor are not changed, the packer will consider itself is under monitoring and exit early.

The big challenge to this experiment is that we do not have the ground truth (e.g., source code or the binary code with no packer) for most samples. We apply two statistic measures from the previous work to assess whether BinUnpack can recover the original code: entropy deviation [55] and "code-to-data" ratio [86]. The byte entropy value examines the randomness in binary code, and it has been used to efficiently recognize packed binary [4, 54, 55, 74]. If the sample is being compressed or encrypted, the entropy value is typically high. According to PinDemonium's evaluation [55], a entropy deviation value of 0.4 between the unpacked version and the packed version is sufficient to verify the correct process dump. Another measure comes from Eureka [86], as the code-to-data ratio would increase when a malware sample unpacks itself. Eureka uses the threshold of 0.5 to determine the end of unpacking.

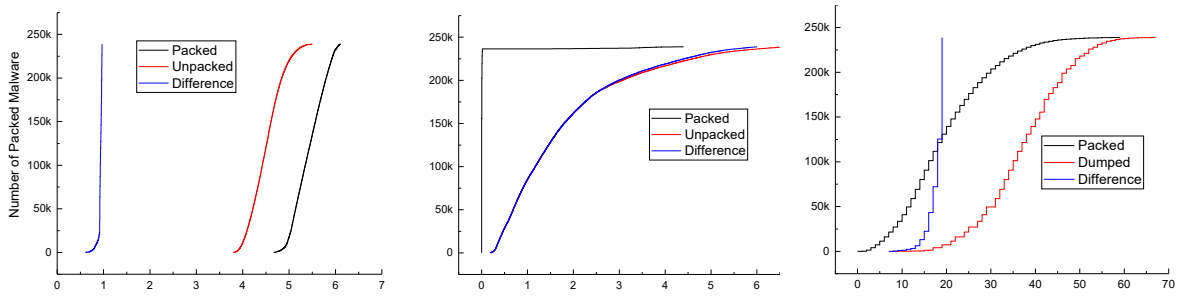
We did not select these two measures by accident. They are complementary to each other, because the attempts to lower the entropy value will increase code-to-data ratio eventually [99]. As shown in Figure 8(a), all of the entropy deviations are beyond the threshold 0.4. The deviation value is from 0.61 to 0.96 with the average value 0.93. For "code-to-data" ratio evaluation (Figure 8(b)), 99.1% of packed malware's ratio ranges from 0.0 to 0.03, and the remaining 0.9%'s ratio is from 1.1 to 4.5, which is far above the threshold 0.5. We further study these outliers and find that they are packed by Armadillo or SoftwarePassport. Unlike other packers, these two packers did not apply any instruction-level obfuscation. Instead, they use double processes to mislead unpackers. However, these outlier packed samples still exhibit a high entropy value. Figure 8(c) shows the VirusTotal detection numbers for the packed malware, the outputs of BinUnpack, and their differences, respectively. After BinUnpack's preprocessing, there are 7 to 19 extra anti-virus scanners (the average number is 18) are able to recognize that malware. Our evaluation demonstrates the effectiveness of BinUnpack against wild packed malware.

9 RELATED WORK

Several previous work has confirmed that erasing the IAT of original code is common in packers [17, 44, 82, 86, 97]. However, the work to utilize IAT rebuilding for unpacking and resist to various evasions at the same time is rare. We have discussed the status quo of generic unpacking techniques in Section 2 and compared the recent tools in Section 8.1. This section focuses on other related work.

The latest talk [100] raises three concerns on malware unpacking, and they fits our motivations perfectly: 1) researchers often underestimate the complexity of packers; 2) anti-virus products often report false alarms when scanning packed benign programs; 3) the pervasive packed malware has severely limited the accuracy of machine learning. Therefore, advanced generic unpacking is desperately necessary. The recent work studies the common ways used by malware to detect the existence of a DBI tool and develops an anti-DBI resistant unpacker [76]. However, the overhead is still quite high, and it cannot handle the process hollowing technique. Xabier et al. customize multi-path exploration techniques on packed code [98]. The purpose is to trigger the unpacking routine that checks runtime environment. We believe BinUnpack can also benefit from multi-path exploration to execute the hidden unpacking routine. Debray et al. try to extract the unpacking routine code instead of original code [18], which can provide insight to the mechanism of custom packers.

Another related direction is DLL hijacking and its prevention. DLL hijacking [79] is originally designed for malicious component loading, and several methods have been proposed to prevent DLL hijacking. Taeho Kwon and Zhendong Su [46, 47] present an automatic technique to detect unsafe component loadings. Their tool profiles an application's dynamic loading behaviour via hooking the APIs that are used to load component. These APIs include "LdrLoadDLL", "LdrpLoadDll", and "LdrpMapDll", which we have introduced in the Figure 10. Byungho Min and Vijay Varadharajan [61, 62] propose a cross verification mechanism for secure execution and dynamic component loading. They also hook the APIs including "LdrLoadDLL" to monitor the dynamic loading behaviour. All of these DLL hijacking prevention work assume that DLL hijackers can not enter the OS kernel. In contrast, our work studies how to use DLL hijacking for defense purpose, and the available resources for defenders are rich. For example, BinUnpack can enter



(a) Entropy Value (b) Code-to-Data Ratio (c) VirusTotal Detection Number

Figure 8: The cumulative distribution result of unpacking wild packed malware.

the OS kernel to hook native APIs. Therefore, using these DLL hijacking prevention mechanisms to defeat BinUnpack becomes increasingly difficult.

10 DISCUSSION & CONCLUSION

The prototype of BinUnpack has several limitations. First, we find some custom packers can evade BinUnpack by detecting the movement of mouse cursor. Our next work is to create an artificial, realistic environment that can simulate how users interact with OS. Like other work [44, 49, 55], we find some unpacked code can not function correctly. The reason is IAT obfuscation such as API redirection [39] renders the rebuilt IAT incomplete. Our current design only needs to make sure that the rebuilt IAT is different from unpacking routine IAT. However, accurately reconstructing the whole IAT requires heavyweight data flow analysis. We leave it as our future work. Another interesting direction is to study the feasibility of applying BinUnpack's idea to Linux malware [16].

Packed malware in circulation is a tremendous amount. The existing generic unpacking tools are limited by the high overhead and lack of anti-analysis resistance. In this paper, we develop a novel unpacking approach, BinUnpack, which is based on capturing the "rebuilt-then-called" feature instead of "written-then-executed" memory. BinUnpack's design is free from tedious memory access tracing and results in very small runtime overhead. To withstand anti-hooking tricks, we develop BinUnpack's API monitor module by kernel-level DLL hijacking. Our large-scale experiments demonstrate the efficacy and generality of BinUnpack.

11 ACKNOWLEDGMENTS

We thank the ACM CCS 2018 anonymous reviewers for their valuable feedback. This research was supported in part by the Natural Science Foundation of Hubei Province of China grant 2017CFB307, the National Natural Science Foundation of China grants (U1636107, 61373168, 61332019, and 61672394), the National Key R&D Program of China (2017YF-B0802903), Project 2117H14243A, and Sichuan Province Research and Technology Supporting Plan. Jiang Ming was also supported by UT System STARs Program.

REFERENCES

[1] Aguila. 2016. Scylla - x64/x86 Imports Reconstruction. https://github.com/NtQuery/Scylla. (2016).
[2] Anonymous Author. 2004. Hooking Windows API - Technics of hooking API functions on Windows. *The CodeBreakers Journal* 1, 2 (2004).
[3] Piotr Bania. 2009. Generic Unpacking of Self-modifying, Aggressive, Packed Binary Programs. https://arxiv.org/abs/0905.4581. (2009).
[4] Munkhbayar Bat-Erdene, Taebeom Kim, Hyundo Park, and Heejo Lee. 2017. Packer Detection for Multi-Layer Executables Using Entropy Analysis. *Entropy* 19, 3 (2017).
[5] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. 2009. Scalable, Behavior-Based Malware Clustering. In *Proceedings of the 16th Network and Distributed System Security Symposium (NDSS'09)*.
[6] Henry Belot and Stephanie Borys. 2017. Ransomware attack still looms in Australia as Government warns WannaCry threat not over. http://www.abc.net.au/news/2017-05-15/ransomware-attack-to-hit-victims-in-australia-government-says/8526346. (May 16 2017).
[7] Leyla Bilge, Andrea Lanzi, and Davide Balzarotti. 2011. Thwarting Real-time Dynamic Unpacking. In *Proceedings of the Fourth European Workshop on System Security (EUROSEC'11)*.
[8] Lutz Böhne. 2008. Pandora's Bochs: Automatic unpacking of malware. *University of Mannheim* 6 (2008).
[9] Guillaume Bonfante, Jose Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier, and Aurélien Thierry. 2015. 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)*.
[10] Denis Bueno, Kevin J. Compton, Karem A. Sakallah, and Michael Bailey. 2013. Detecting Traditional Packers, Decisively. In *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'13)*.
[11] Alexei Bulazel and Bülent Yener. 2017. A Survey On Automated Dynamic Malware Analysis Evasion and Counter-Evasion: PC, Mobile, and Web. In *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*.
[12] Jamie Butler and Kris Kendal. 2007. Blackout: What really happened. Black Hat USA. (2007).
[13] Joan Calvet and Pierre-Marc Bureau. 2010. Understanding Swizzor's Obfuscation Scheme. REcon 2010. (2010).
[14] Joan Calvet, Fanny Lalonde Lévesque, Jose M. Fernandez, Erwann Traourouder, Francois Menet, and Jean-Yves Marion. 2015. WaveAtlas: surfing through the landscape of current malware packers. Virus Bulletin Conference. (2015).
[15] Cisco. 2017. Cisco 2017 Midyear Cybersecurity Report. https://www.cisco.com/c/m/en_au/products/security/offers/annual-cybersecurity-report-2017.html. (2017).
[16] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. 2018. Understanding Linux Malware. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P'18)*.
[17] DataRescue. 2005. Using the Universal PE Unpacker Plug-in included in IDA Pro 4.9 to unpack compressed executables. https://www.hex-rays.com/products/ida/support/tutorials/unpack_pe/unpacking.pdf. (2005).
[18] Saumya Debray and Jay Patel. 2010. Reverse Engineering Self-Modifying Code: Unpacker Extraction. In *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE'10)*.
[19] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*.
[20] Ken Dunham and Egan Hadsell. 2011. Malcode Context of API Abuse. SANS Institute InfoSec Reading Room. (2011).
[21] Peter Ferrie. 2008. Anti-unpacker tricks. Virus Bulletin. (2008).
[22] Stephen Fewer. 2013. Reflective DLL Injection. https://github.com/stephenfewer/ReflectiveDLLInjection. (2013).

[23] Halvar Flake. 2004. Structural comparison of executable objects.. In *Proceedings of the 2004 GI International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA'04)*.

[24] Jianming Fu, Xinwen Liu, and Binling Cheng. 2011. Malware behavior Capturing based on Taint Propagation and Stack Backtracing. In *Proceedings of the 10th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom'11)*.

[25] Gabriela Limon Garcia. 2007. Forensic physical memory analysis: an overview of tools and techniques. In *TKK T-110.5290 Seminar on Network Security*. 305–320.

[26] Joseph Gardiner, Marco Cova, and Shishir Nagaraja. 2014. Command & Control: Understanding, Denying and Detecting. *arXiv CoRR* abs/1408.1136 (2014). http://arxiv.org/abs/1408.1136

[27] Emre Güler. 2017. Anti-Sandboxing Techniques in Cerber Ransomware. VMRay Blog. (2017).

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

[29] Irfan Ul Haq, Sergio Chica, Juan Caballero, and Somesh Jha. 2017. Malware Lineage in the Wild. arXiv:1710.05202 [cs.CR]. (2017).

[30] Ashkan Hosseini. 2017. Ten Process Injection Techniques: A Technical Survey of Common and Trending Process Injection Techniques. Endpoint Security Blog. (2017).

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

[32] Xin Hu, Tzi cker Chiueh, and Kang G. Shin. 2009. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and Communications Security (CCS'09)*.

[33] Galen Hunt and Doug Brubacher. 1999. Detours: Binary interception of win32 functions. In *3rd Usenix Windows NT Symposium*.

[34] Huorong Network. 2017. The Introduction to Huorong Anti-Virus Engine. http://www.huorong.cn/doc/introduce_engine.pdf. (2017).

[35] Ryoichi Isawa, Masakatu Morii, and Daisuke Inoue. 2016. Comparing Malware Samples for Unpacking: A Feasibility Study. In *Proceedings of the 11th Asia Joint Conference on Information Security*.

[36] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. 2007. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 5th ACM Workshop on Recurring Malcode (WORM'07)*.

[37] Yuhei Kawakoya, Makoto Iwamura, and Mitsutaka Itoh. 2010. Memory behavior-based automatic malware unpacking in stealth debugging environment. In *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE'10)*.

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

[39] Yuhei Kawakoya, Eitaro Shioji, Yuto Otsuki, Makoto Iwamura, and Takeshi Yada. 2017. 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)*.

[40] Hyung Chan Kim, Tatsunori ORII, Katsunari Yoshioka, Daisuke Inoue, Jungsuk Song, Masashi ETO, Junji Shikata, Tsutomu Matsumoto, and Koji Nakao. 2011. An Empirical Evaluation of an Unpacking Method Implemented with Dynamic Binary Instrumentation. *IEICE TRANSACTIONS on Information and Systems* E94-D, 9 (2011).

[41] Paul Kimayong. 2017. New Breed of Cerber Ransomware Employs Anti-Sandbox Armoring. https://www.cyphort.com/new-breed-of-cerber-ransomware-employs-anti-sandbox-armoring. (2017).

[42] Dhilung Kirat and Giovanni Vigna. 2015. MalGene: Automatic Extraction of Malware Analysis Evasion Signature. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*.

[43] Eugene Kolodenker, William Koch, Gianluca Stringhini, and Manuel Egele. 2017. PayBreak: Defense Against Cryptographic Ransomware. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS'17)*.

[44] David Korczynski. 2016. 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)*.

[45] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. 2004. Static Disassembly of Obfuscated Binaries. In *Proceedings of the 13th USENIX Security Symposium (USENIX Security'04)*.

[46] Taeho Kwon and Zhendong Su. 2010. Automatic Detection of Unsafe Component Loadings. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA'10)*.

[47] Taeho Kwon and Zhendong Su. 2012. Automatic detection of unsafe dynamic component loadings. *IEEE Transactions on Software Engineering* 38, 2 (2012), 293–313.

[48] John Leitch. 2011. Process Hollowing. https://www.autosectools.com/Process-Hollowing.pdf. (2011).

[49] Julien Lenoir. 2015. Implementing your own generic unpacker. HITB Singapore 2015. (2015).

[50] Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. 2012. Lines of Malicious Code: Insights into the Malicious Software Industry. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*.

[51] Limin Liu, Jiang Ming, Zhi Wang, Debin Gao, and Chunfu Jia. 2009. Denial-of-Service Attacks on Host-Based Generic Unpackers. In *Proceedings of the 11th International Conference on Information and Communications Security (ICICS'09)*.

[52] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. 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)*.

[53] Gustav Lundsgård and Victor Nedström. 2016. *Bypassing modern sandbox technologies*. Master's thesis. Lund University.

[54] Robert Lyda and James Hamrock. 2007. Using Entropy Analysis to Find Encrypted and Packed Malware. *IEEE Security and Privacy* 5, 2 (2007).

[55] Sebastiano Mariani, Lorenzo Fontana, Fabio Gritti, and Stefano D'Alessio. 2016. PinDemonium: a DBI-based generic unpacker for Windows executables. Black Hat USA. (2016).

[56] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. 2007. OmniUnpack: Fast, generic, and safe unpacking of malware. In *Proceedings of the 23nd Annual Computer Security Applications Conference (ACSAC'07)*.

[57] Aldo Mazzeo. 2016. Custom LoadLibrary implementation. https://github.com/gbmaster/loadLibrary. (2016).

[58] Microsoft. last reviewed, 05/08/2018. Linking Explicit. https://msdn.microsoft.com/en-us/library/784bt7z7.aspx. (last reviewed, 05/08/2018).

[59] Microsoft. last reviewed, 05/08/2018. Linking Implicitly. https://msdn.microsoft.com/en-us/library/d14wsce5.aspx/. (last reviewed, 05/08/2018).

[60] Matt Miller. 2008. Using dual-mappings to evade automated unpackers. http://www.uninformed.org/?v=10&a=1. (2008).

[61] Byungho Min and Vijay Varadharajan. 2015. Secure Dynamic Software Loading and Execution Using Cross Component Verification. In *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'15)*.

[62] Byungho Min and Vijay Varadharajan. 2016. Rethinking Software Component Security: Software Component Level Integrity and Cross Verification. *Comput. J.* 59, 11 (2016), 1735–1748.

[63] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. 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)*.

[64] M. Morgenstern and A. Marx. 2008. Runtime Packer Testing Experiences. 2nd International CARO Workshop. (2008).

[65] Maik Morgenstern and Hendrik Pilz. 2010. Useful and useless statistics about viruses and anti-virus programs. 4th International CARO Workshop. (2010).

[66] Ellen Nakashima and Philip Rucker. 2017. U.S. declares North Korea carried out massive WannaCry cyberattack. The Washington Post. (December 19 2017).

[67] Lakshman Nataraj. 2013. Nearly 70% of Packed Windows System files are labeled as Malware. UCSB Sarvam Blog. (2013).

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

[69] Jon Oberheide, Michael Bailey, and Farnam Jahanian. 2009. PolyPack: An Automated Online Packing Service for Optimal Antivirus Evasion. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies (WOOT'09)*.

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

[71] Digit Oktavianto and Iqbal Muhardianto. 2013. *Cuckoo Malware Analysis: Analyze malware using Cuckoo Sandbox*. Packt Publishing Ltd.

[72] Oreans Technologies. last reviewed, 05/08/2018. Themida: Advanced Windows Software Protection System. https://www.oreans.com/themida.php. (last reviewed, 05/08/2018).

[73] Panda Security. 2017. PandaLabs Annual Report 2017. https://www.pandasecurity.com/mediacenter/src/uploads/2017/11/PandaLabs_Annual_Report_2017.pdf. (2017).

[74] Roberto Perdisci, Andrea Lanzi, and Wenke Lee. 2008. Classification of Packed Executables for Accurate Computer Virus Detection. *Pattern Recognition Letters* 29, 14 (Oct. 2008).

[75] I Phillips. 2006. Windows Vista security: first impressions. *information security technical report* 11, 4 (2006), 176–185.

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

[77] Danny Quist and Valsmith. 2007. Covert Debuging: Circumventing Software Armoring Techniques. Black Hat USA. (2007).

[78] Jason Raber and Brian Krumheuer. 2009. QuietRIATT: Rebuilding the Import Address Table Using Hooked DLL Calls. Black Hat DC. (2009).
[79] Max Rival. last reviewed, 05/08/2018. Dynamic-Link Library Hijacking. https://www.exploit-db.com/docs/31687.pdf. (last reviewed, 05/08/2018).
[80] J Robbins. 1999. Debugging Windows based applications using WinDbg. *Miscrosoft Systems Journal* (1999).
[81] Kevin A. Roundy and Barton P. Miller. 2010. Hybrid Analysis and Control of Malware. In *Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection (RAID'10)*.
[82] Kevin A. Roundy and Barton P. Miller. 2013. Binary-code Obfuscations in Prevalent Packer Tools. *Comput. Surveys* 46, 1 (2013).
[83] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. 2006. PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*.
[84] Mark E Russinovich, David A Solomon, and Alex Ionescu. 2012. *Windows Internals (6th Edition)*. Microsoft Press.
[85] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. 2009. Automatic reverse engineering of malware emulators. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P'09)*.
[86] Monirul Sharif, Vinod Yegneswaran, Hassen Saidi, Phillip Porras, and Wenke Lee. 2008. Eureka: A framework for enabling static malware analysis. In *Proceedings of the 13th European Symposium on Research in Computer Security (ESORICS'08)*.
[87] Paria Shirani, Lingyu Wang, and Mourad Debbabi. 2017. BinShape: Scalable and Robust Binary Library Function Identification Using Function Shape. In *Proceedings of the 14th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'17)*.
[88] Michael Sikorski and Andrew Honig. 2012. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press.
[89] Alexey Sintsov. 2010. Writing JIT-spray shellcode for fun and profit. DigitĐřl Security Research Group (DSecRG). (2010).
[90] Sherri Sparks and Jamie Butler. 2005. Shadow Walker: Raising The Bar For Windows Rootkit Detection. Black Hat Japan. (2005).
[91] Joe Stewart. 2007. Unpacking with OllyBonE. http://www.joestewart.org/ollybone/. (2007).
[92] Hung Min Sun, Yue Hsun Lin, and Ming Fung Wu. 2006. API Monitoring System for Defeating Worms and Exploits in MS-Windows System. In *Proceedings of the 11th Australasian Conference on Information Security and Privacy (ACISP'06)*.
[93] Gabor Szappanos. 2007. Exepacker blacklisting. Virus Bulletin. (2007).
[94] Brad Taylor. 2017. Extortion-based cyber attacks: The next evolution in profit-motivated attack strategies. https://www.helpnetsecurity.com/2017/11/09/extortion-based-cyber-attacks/. (2017).
[95] Check Point Research Team. 2017. May's Most Wanted Malware: Fireball and Wannacry Impact More Than 1 in 4 Organizations Globally. https://blog.checkpoint.com/2017/06/20/mays-wanted-malware-fireball-wannacry-impact-1-4-organizations-globally. (2017).
[96] Check Point Research Team. 2018. March's Most Wanted Malware: Cryptomining Malware That Works Even Outside the Web Browser on the Rise. https://blog.checkpoint.com/2018/04/13/marchs-wanted-malware-cryptomining-malware-works-even-outside-web-browser-rise. (2018).
[97] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. 2015. 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)*.
[98] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G. Bringas. 2016. RAMBO: Run-time packer Analysis with Multiple Branch Observation. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'16)*.
[99] Xabier Ugarte-Pedrero, Igor Santos, Borja Sanz, Carlos Laorden, and Pablo Garcia Bringas. 2012. Countering entropy measure attacks on packed software detection. In *Proceedings of the 2012 IEEE Consumer Communications and Networking Conference (CCNC'12)*.
[100] Giovanni Vigna and Davide Balzarotti. 2018. When Malware is Packing Heat. In *USENIX Enigma 2018*.
[101] Carsten Willems, Thorsten Holz, and Felix Freiling. 2007. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security & Privacy* 5, 2 (2007).
[102] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A generic approach to automatic deobfuscation of executable code. In *Proceedings of the 36th IEEE Symposium on Security & Privacy (S&P'15)*.
[103] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. 2012. V2E: Combining Hardware Virtualization and Softwareemulation for Transparent and Extensible Malware Analysis. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE'12)*.
[104] Wei Yan, Zheng Zhang, and Nirwan Ansari. 2008. Revealing Packed Malware. *IEEE Security and Privacy* 6, 5 (Sept. 2008).
[105] Udi Yavo and Tomer Bitton. 2016. Captain Hook: Pirating AVS to Bypass Exploit Mitigations. Black Hat USA. (2016).
[106] Lenny Zeltser. 2017. The History of Fileless Malware – Looking Beyond the Buzzword. https://zeltser.com/fileless-malware-beyond-buzzword/. (2017).

Algorithm 2 Algorithm of MyNtMapViewOfSection.

SectionHandle: a handle to a section object which will be mapped into memory.
**BaseAddress*: a pointer to a base address where the section will be mapped to.

```
1:  function MyNtMapViewOfSection(SectionHandle, ... *BaseAddress, ...)
      /* Resolve the FileName from SectionHandle */
2:    FileName ← ResolveFileName(SectionHandle)
3:    if FileName = "c:\windows\system32\kernel32.dll" then
        /* Map the home-made DLL "MyKernel32.dll"
        into memory address of mapped_address */
4:        mapped_address ← MapFile(MyKernel32.dll)
        /* set "*BaseAddress" to "mapped_address" */
5:        *BaseAddress ← mapped_address
        /* Let "LdrpMapDll" function to reload the
        home-made kernel32.dll */
6:        return STATUS_IMAGE_NOT_AT_BASE
7:    else
8:        return NtMapViewOfSection(SectionHandle,...
              *BaseAddress,...)
9:    end if
10: end function
```

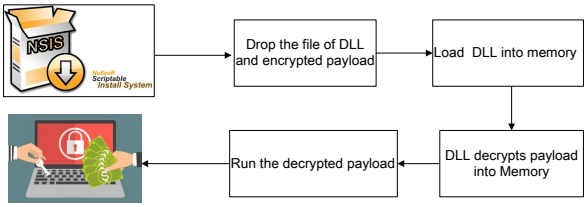


Figure 11: The typical attacking procedure of NSIS packed ransomware.

Table 7: The results of hook-evasion resistance evaluation. "✓" indicates the API monitor/sandbox is resilient to that evasion type.

Packers	Evasion Type	Detours	CWSandbox	Cuckoo	BinUnpack
Armadillo	Child process		✓	✓	✓
Pespin	Child process		✓	✓	✓
Asprotect	Stolen code				✓
Pelock	Stolen code				✓
Yoda's Protector	Stolen code				✓
Yoda's Crypter	Stolen code				✓
PEP	Stolen code				✓
Enigma	Stolen code				✓
Themida	Stolen code				✓
Cerber's packer	Process hollowing		✓	✓	✓
Cerber's packer	Crash hooking module				✓
Custom UPX	Custom loader [57]				✓
Custom UPX	Stealth loader [39]				✓
Custom UPX	DLL integrity check				✓

Table 5: Standard path search order.

Order
The directory of the application loaded
The system directory
The 16-bit system directory
The Windows directory
The current directory
The PATH environment variable

APPENDIX

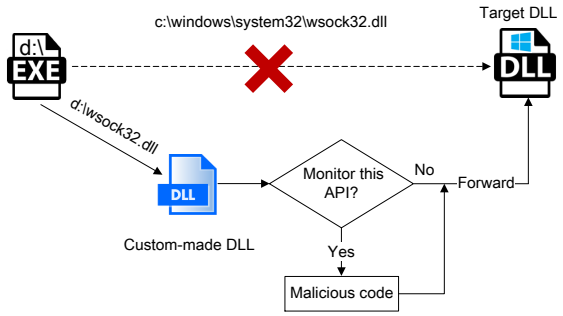


Figure 9: Hijacking Windows network management DLL, wsock32.dll, with a custom-made DLL.

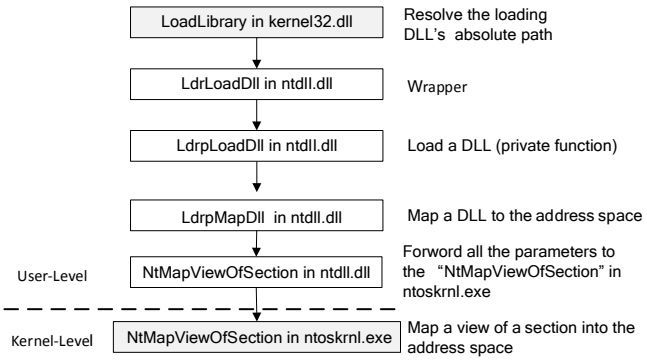


Figure 10: The call chain of LoadLibrary. LoadLibrary first resolves the loading DLL's absolute path, including reading core DLL's path from the particular Registry key. Then, LoadLibrary forwards the loading DLL's full path to the next level of API, and it finally invokes NtMapViewOfSection.

Table 6: The overhead BinUnpack introduces when working with benign programs.

Sample	Benign (ms)	BinUnpack (ms)			Relative Slowdown
		IATComparison	OEPSearch	Dump	
tasklist	109	0.24	0	0	0.22%
winrar	10624	30.57	0	0	0.29%
WinPcap	3620	0.45	0	0	0.01%
IE	254	0.65	0	0	0.26%
FireFox	231	1.97	0	0	0.85%
Chrome	161	2.39	0	0	1.48%