PAPER

BareUnpack: Generic Unpacking on the Bare-Metal Operating System

Binlin CHENG[†,††], ***Member and*** **Pengwei LI**[†††a)], ***Nonmember***

SUMMARY Malware has become a growing threat as malware writers have learned that signature-based detectors can be easily evaded by packing the malware. Packing is a major challenge to malware analysis. The generic unpacking approach is the major solution to the threat of packed malware, and it is based on the intrinsic nature of the execution of packed executables. That is, the original code should be extracted in memory and get executed at run-time. The existing generic unpacking approaches need a simulated environment to monitor the executing of the packed executables. Unfortunately, the simulated environment is easily detected by the environment-sensitive packers. It makes the existing generic unpacking approaches easily evaded by the packer. In this paper, we propose a novel unpacking approach, BareUnpack, to monitor the execution of the packed executables on the bare-metal operating system, and then extracts the hidden code of the executable. BareUnpack does not need any simulated environment (debugger, emulator or VM), and it works on the bare-metal operating system directly. Our experimental results show that BareUnpack can resist the environment-sensitive packers, and improve the unpacking effectiveness, which outperforms other existing unpacking approaches.

key words: *malware analysis, environment-sensitive techniques, simulated environment, generic unpacker*

1. Introduction

Malicious software (malware) development has become a booming underground market. Driven by the rich profit, cyber-criminals are highly motivated to undermine malware detection/analysis by applying numerous obfuscation schemes. Instead of obfuscating the malware directly, malware writers heavily rely on the packer, which is a program that "packing" the malware in layers of compression or encryption so that it has a different looking than the original one, to evade detection of signature-based detection. It is reported that over 85% of malware is packed [1], [2]. The packed malwares seriously degrade the effectiveness of signature-based detection.

Generic unpacking is a promising solution to the problem of unpacking the packed executables as it does not rely on signatures. Regardless of which packer might be applied, the original code must be present in memory and then be executed. Based on this intrinsic nature of packed executables, one could extract the hidden code at run-time as a raw memory dump.

The problem

However, the existing generic unpacking approaches may not be reliable in all cases and can be easily evaded. These approaches are all based on a simulated environment to capture the "written-then-executed" behaviors of packed executables [3]–[5]. Unfortunately, the simulated environment will increase the risk that exposes the existence of unpacking approaches. Nowadays, the packer writer produced more complex packers which evade the existing generic unpacking approaches via detecting the simulated environment [6], [7]. For example, the previous works have presented many techniques to check the existence of simulated environment, including anti-debugging, and anti-emulating [2], [8]. When packed executables recognize the existence of the simulated environment, they will stop execution immediately.

The challenge

The simulated environment usually has many features which are different from the bare-metal OS (operating system). As it is difficult to construct the simulated environment which is exactly the same to the bare-metal OS, we aim to answer the question: "can we design a generic unpacking approach which not relies on the simulated environment, and on the bare-metal OS directly ?" If we can design a generic unpacking approach that on the bare-metal OS directly, it can help us to resist the environment-sensitive packers and improve the unpacking effectiveness.

Our approach

In this paper, we present a generic unpacking approach, called BareUnpack. BareUnpack is working on the fully bare-metal OS, not relies on any simulated environment (debugging, virtual machine, or emulation). Compared to the existing works, BareUnpack is not easily evaded by the environment-sensitive techniques. BareUnpack's design principle is motivated by a basic practice at the execution of packed executables. During the execution, the packed executables will restore the import address table (IAT) before calling the API from the original code. In the paper, we plan to give the introduction that how to monitor the execution of the packed executables on the bare-metal OS and propose BareUnpack, a generic unpacking approach running on the bare-metal OS.

Contributions

The main contributions of this paper are as follows:

Manuscript received December 28, 2017.
Manuscript revised July 6, 2018.
Manuscript publicized September 12, 2018.

[†]The author is with the Faculty School of Computer Science, Hubei Normal University, China.
[††]The author is with the Faculty College of Arts and Science, Hubei Normal University, China.
[†††]The author is with School of Technology, Nanjing Audit University, China.
a) E-mail: pwli@nau.edu.cn (Corresponding author)

DOI: 10.1587/transinf.2017EDP7424

- We propose a novel hooking method, "packed IAT hooking", which can hook the IAT of packed executables to monitor the behavior of it on the bare-metal OS;
- We propose BareUnpack, a generic unpacking approach running on the fully bare-metal OS environment via packed IAT hooking;
- We have evaluated BareUnpack on real-world packers. Our experiments demonstrate that BareUnpack achieves more effective unpack results than the existing approaches which rely on the simulated environment.

Organization

The rest of this paper is organized as follows. Section 2 provides the background. Section 3 introduces our approach, BareUnpack. Section 4 presents the evaluation. Section 5 describes related work. Section 6 concludes.

2. Background and Motivation

We now introduce the background in this section.

2.1 Generic Unpacking

A packer is a program that compresses or encrypts an executable file into a new file. When the new file is executed, the unpacking stub of the packed file will restore the original code and transfer the control flow to the original code.

By taking advantage of the intrinsic nature of packed executables, that is, the original code must be written in memory pages and get executed at execution, the generic unpacking approach monitors the behaviors of "written-then-executed" at execution, and exact the original code as a raw memory dump.

Thus, these unpacking approaches need to monitor program execution and memory writes in the simulated environment, determine whether the code at execution is newly generated, if so, they extract the newly code immediately.

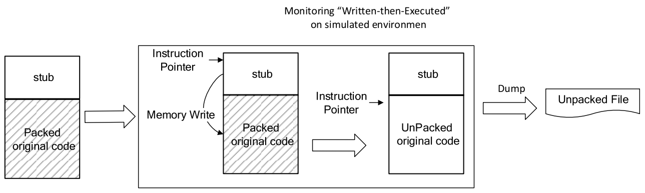

Fig. 1 The process of the existing generic unpacking approaches.

Table 1 Environment sensitive techniques.

Packing Tools	**Checking Type**	**Checking Point**
Shrinker	API Check	KiUserExceptionDispatcher()
MSLRH	API Check	NtQueryInformationProcess()
Yoda's Protector	API Check	SuspendThread()
ExeCryptor	Memory Check	PEB Structure
Yoda's Crypter	Memory Check	Section table
HyperUnpackMe2	Memory Check	EPROCESS Structure
Obsidium	Memory Check	Check encrpted section
Pelock	Multiple Thread	Create Multiple Thread
PECompact	API Check&Memory Check	Section table
Armadillo	API Check&Memory Check	Checking Hardware
Themida	Sandbox Check	

Figure 1 illustrates the process of the existing generic unpacking approaches.

2.2 Environment Sensitive Packer

The generic unpacking approach is the main solution to the threat of packed malware. However, the main shortcoming of generic unpacking approach is that packed malware can be aware of the simulated environment, then the packed malware can mislead the unpacking approaches to executing a benign execution path. For example, Table 1 shows several anti-unpacking techniques which are used to thwart unpacking approach when there is an emulator or a debugger running. The CPU bugs, alignment checking, and registers are also could be used to detect the simulated environments [9], [10].

Once the packed malware is aware of the existence of the simulated environment, it will stop execution immediately to mislead the generic unpacking approaches.

3. Overview

We will describe the basic idea of our approach in this section.

The existing generic unpacking approaches are based on the simulated environments to capture the "written-then-executed" behaviors of packed executables [1], [4], [5]. Their key motivation is that, when the packed executable executes the hidden code which generated at execution, it indicates that the packed executable has restored the original code. As the simulated environment is easy to be detect by the packed malware, which causes the existing generic unpacking approach fail to unpack. We aim to find a new generic unpacking approach which does not rely on the simulated environment, and on the bare-metal OS directly.

3.1 Execution Monitor

To monitor a given application's execution on the bare-metal OS, the most used way is API hooking, intercepting a call to a function. Although many alternatives are available to hook API call [11], [12], all of them cannot completely fulfill our requirements. In the following, we first summarize existing hooking in detail. Then we describe the procedure of loading IAT of packed executables. At last, we propose a novel hooking method suitable to hook the IAT of packed executables.

Existing API hooking methods

When an application calls a function, the API hooking reroutes the control flow to a different location where the hooking function resides. The hooking function then performs its own operations and transfers control flow back to the original API function.

An API call typically goes through three key data structures of Windows OS: IAT (Import Address Table), EAT (Export Address Table), and SSDT (System Service Dispatch Table). All of these three tables act as lookup tables to

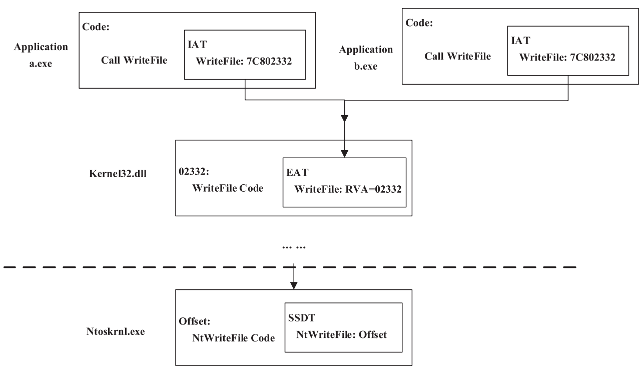

Fig. 2 The execution path of windows API.

preserve function address. Depending on which table manipulated, API hooking can be divided into IAT hooking, EAT hooking, and SSDT hooking.

Figure 2 shows the procedure of API call by showing two applications (“a.exe” and “b.exe”) call the API “WriteFile”. The procedure can be illustrated as followers:

1. The application “a.exe” looks up the address of “WriteFile” in its own IAT (Import Address Table)

2. IAT directs the function call to the actual memory address of “WriteFile” located in the EAT (Export Address Table) of Kernel32.dll.

3. In the kernel mode, KiSystemSerivce (kernel mode interrupt handler) looks up the address of the “NTWriteFile” in the SSDT (System Service Dispatch Table) and calls “NTWriteFile” on behalf of the application “a.exe”.

From the procedure above we can see that the API call of “WriteFile” will go through three key data structures in the Windows OS: IAT (Import Address Table), EAT (Export Address Table) and SSDT (System Service Dispatch Table). The common feature of these data structures is that they are the tables which preserve the address of the function. If a program replaces the address of “WriteFile” API in any one of these tables, then it can intercept the call to the API “WriteFile”. Depending on which table replaced, the API hooking can be divided into different classes: IAT hooking, EAT hooking and SSDT hooking.

In addition to the common features, the various hooking methods also have their own features. IAT hooking can hook all the APIs used for the given process which holds the IAT. While both EAT hooking and SSDT hooking do not know the which APIs used for the given process (shown as Table 2).

BareUnpack needs to monitor the execution of the given process (packed executable). If we use the EAT hooking or SSDT hooking, we need to hook all the API in the EAT or the SSDT as the effective range. Too many hooking will lead to the following limitations:

1. It is easy to be aware by the environment sensitive packer;

Table 2 Effective range of different API hooking methods.

Hooking Methods	Process-Special
IAT hooking	✓
EAT hooking	
SSDT hooking	

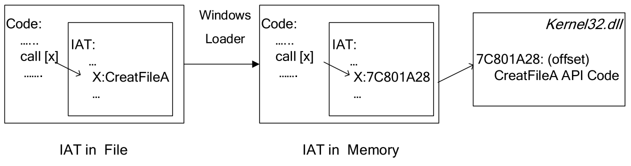

Fig. 3 Standard IAT Loading.

2. It will increase the performance overhead of the bare-metal OS.

Since these limitations, the IAT hooking is more suitable for BareUnpack compared to the EAT hooking and SSDT hooking. In the following, we will introduce the IAT hooking in detail and study whether it can be used for BareUnpack in practice.

IAT Hooking

IAT (Import Address Table) hooking [13], [14] is a widely used hooking method. To understand the IAT hooking, we need understand the IAT first.

IAT is an important data structure for the executable files in the Windows OS. Each PE executable file (e.g. EXE) in Windows OS contains an IAT, which holds names for functions that need to be imported from an external DLL.

When an executable is loaded, the required DLLs are mapped into the memory address space of the application, and the IAT is filled in by the Windows Loader with the virtual addresses of each imported function. This table (IAT) is referred to by indirect control flow instructions in the program to call the functions in the linked DLL (as shown in Fig. 3).

Having loaded into memory, the executable looks up the address of the function in IAT when needs a function call. We could redirect the function call to a hooking function than the original function via IAT hooking, which rewrites the virtual address in the IAT (as shown in Fig. 4). After the execution of monitoring code, the hooking function can forward the control flow to the original API call.

Restoring IAT

The IAT hooking can be used to monitor the execution of an application effectively. However, it is not suited for the packed executable, as the process of IAT loading of packed executables is different for the normal ones.

If an executable is packed by the packer, the IAT of the executable is erased by the packer to hide the API information of the executable [2], [15]. Since the function names are erased, when loading the packed executable into memory, the Windows Loader cannot implement the operation of converting the function name into the function address as loading the standard IAT loading (shown in Fig. 3). Instead, the packer converts the function name of IAT into the function address by itself, leveraging the API “GetProcAddress”, which converts the function name into the function address. This process is called "Restoring IAT" [2], [16](shown in Fig. 5).

Comparing the Fig. 3 and Fig. 5, we can see that the process of loading IAT for the packed executable (shown as Fig. 5) is different from the standard IAT loading (shown as Fig. 3). As the existing IAT hooking method is based on the process of standard IAT loading, how to hook the IAT of the packed executable is still a problem, and we will study it in the following.

Packed IAT Hooking

In this paper, we propose a novel hooking method to hook the IAT of packed executables. As the process of "Restoring IAT" needs a key API "GetProcAddress", we can hook the API of "GetProcAddress", and replace it with the hooking function "MyGetProcAddress". When the packer tries to call the API "GetProcAddress" to restore the IAT, it will call the hooking function "MyGetProcAddress" instead. Take the API "CreateFileA" for example, "MyGetProcAddress" will convert the function name "CreateFileA" in the IAT into the address of another hooking function of "MyCreateFileA". The hooking function of "MyCreateFileA" will execute the monitoring code and then forward the control flow to the original API call "CreateFileA". In this way, we can hook the IAT of the packed executable. We call this process as "packed IAT hooking" (shown in Fig. 6).

"Packed IAT hooking" is a novel hooking method which we proposed in this paper. Different from the existing hooking methods (SSDT hooking, IAT hooking, and EAT hooking), the "packed IAT hooking" has the following noval features:

1. **Double Hook** "Packed IAT hooking" is a kind of double hooking method. It first hooks the API of "GetProcAddress", replacing it into the hooking function "MyGetProcAddress", and then it hooks the IAT of the packed executable through the function

2. **Penetrating Hook** "Packed IAT hooking" is kind of penetrating hooking method, as it penetrates the pro-

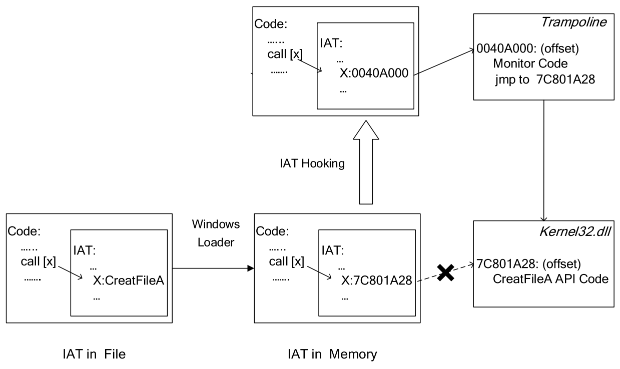

Fig. 4 IAT hooking.

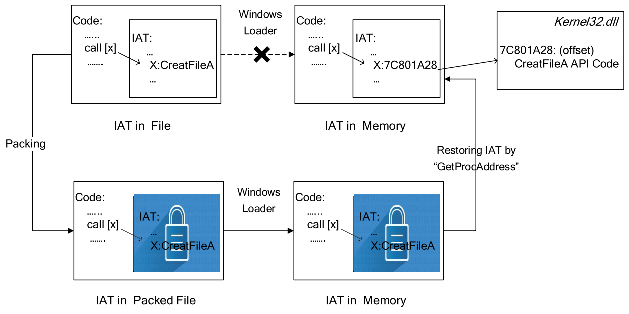

Fig. 5 Restoring IAT.

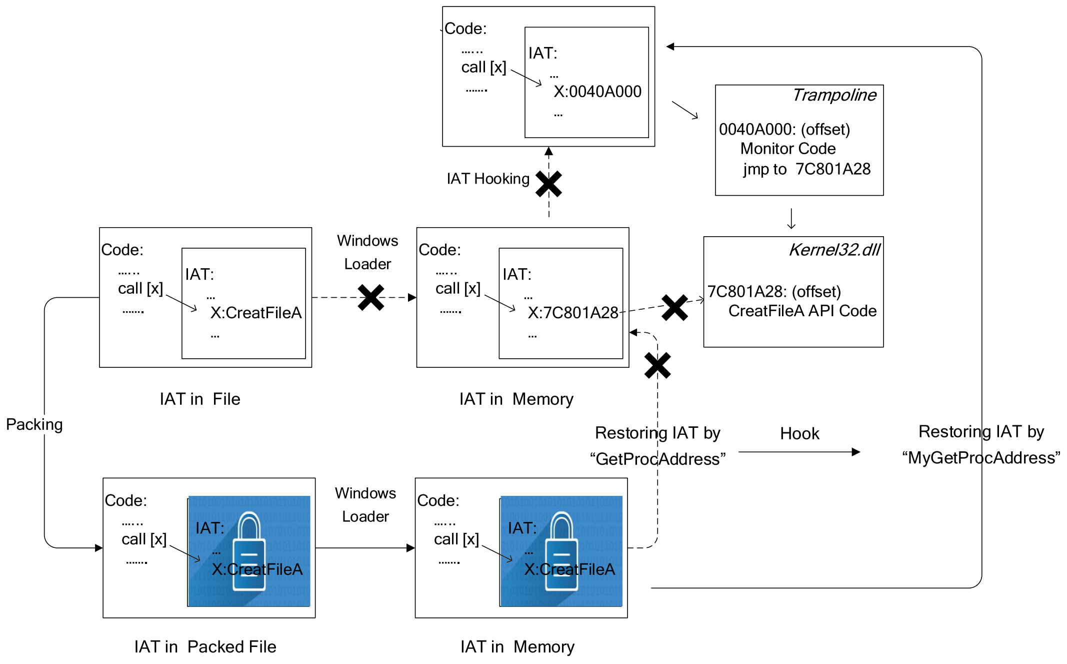

Fig. 6 Packed IAT hooking.

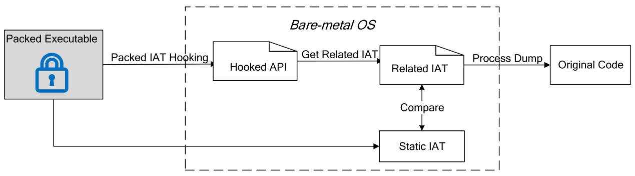

Fig. 7 The framework of BareUnpack.

tecting of packer to hook the IAT of packed executables.

3. **Run-time Hook** "Packed IAT hooking" is kind of run-time hooking method, as it hooks the IAT of the packed executable at the execution of restoring IAT at run-time.

Bare-metal Generic Unpacking

In Sect. 3.1, we have described how to hook the IAT of packed executables via "packed IAT hooking". In this subsection, we will introduce BareUnpack how to work on the bare-metal OS leveraging "packed IAT hooking" method.

Similar to other generic unpacking approaches, our unpacking approach also capture the "written-then-executed" behaviors of packed executables [1], [3]–[5]. That is, when the packed executable executes the hidden code which generated at execution, it indicates that the packed executable has restored the original code. The difference between BareUnpack and other approaches is the running environment. BareUnpack is running on the bare-metal OS, while other approaches are running on the simulated environments.

An overview of BareUnpack is shown in Fig. 7. BareUnpack combines the dynamic analysis and static analysis of the packed executable. In the dynamic analysis, BareUnpack dynamically monitors the execution of the packed executable via "packed IAT hooking"; and examines whether the current API has a related IAT. If so, BareUnpack compares the related IAT with the static IAT which extracted by static analysis. If they are different in terms of IAT's location or content, it means the related IAT is generated at run-time and the original code has been restored. Then, we dump the memory of current process into a file on the disk and exit the execution of the monitored process immediately. The file is the unpacked executable which contains the original code.

The algorithm of BareUnpack is shown in Algorithm 1.

Algorithm 1 The Algorithm of BareUnpack

```
    packedFile: The name of packed file.
 1: function UNPACK(packedFile)
 2:     staticIAT ← StaticAnalysis(packedFile)
 3:     SetPackedIATHooking()
 4:     process ← Execute(packedFile)
 5:     for each $API_i$ in APIMonitor(process) do
 6:         if GetRelatedIAT() $\neq \emptyset$ then
 7:             if relatedIAT() $\neq$ staticIAT then
 8:                 DumpProcess()
 9:                 ExitProcess()
10:             end if
11:         else
12:             Invoke the API handle for process
13:             return
14:         end if
15:     end for
16: end function
```

To identify potential packed code, BareUnpack introduced static analysis. By comparing static analysis results and dynamic analysis results based on "packed IAT hooking". The key idea of "packed IAT hooking" is hooking the API of "GetProcAddress". However, such an approach was limited, considering the API "GetProcAddress" was frequently used both by benign and packed executables. For example, the benign executables may dynamically load API using "GetProcAddress". This may cause false positive to BareUnpack. That is the reason why BareUnpack examines whether the current API has a related IAT before the IAT comparing. If the benign executable dynamically loads API via "GetProcAddress", it has not a related IAT to the loaded API. Then, BareUnpack does not carry out the next IAT comparing. In this way, BareUnpack can preclude the false positive from the benign executables.

While BareUnpack monitors the "written-then-executed" behaviors of packed executables same as the existing generic unpacking approaches, the monitor environment of BareUnpack is different from other approaches. BareUnpack is running on the bare-metal OS via "packed IAT hooking" (shown in Fig. 7), while other approaches are running on simulated environments. The packing IAT hooking enables the BareUnpack monitor the execution of packed executables on the bare-metal OS without any simulated environments.

In the next section, we will evaluate the effectiveness of BareUnpack compared to other approaches.

4. Implementation

As shown in Fig. 6, the key idea of "Packed IAT hooking" is hooking the function of "GetProcAddress". BareUnpack's "GetProcAddress" hooking module is developed on the Microsoft's API hooking framework, Detours[†], containing 120 lines of code in C/C++. We extend Detours to hook "GetProcAddress", use our function "MyGetProcAddress" to replace "GetProcAddress". And "MyGetProcAddress" will convert the function names in the packed IAT into the address of our hooking functions. In this way, we can hook the IAT of the packed executable.

5. Evaluation

In this section, we will evaluate BareUnpack. We conduct our experiments with several objectives in mind. First and foremost, we want to evaluate whether BareUnpack outperforms existing generic unpacking approaches in terms of better effectiveness. To this end, we evaluate BareUnpack and other state-of-the-art generic unpacking approaches against a set of famous known packers. And then

[†]https://www.microsoft.com/en-us/research/project/detours/

Table 3 Comparative evaluation with ground truth dataset.

Packers	PolyUnpack	Renovo	OmniUnpack	CoDisasm	PINdemonium	BareUnpack
NsPack	✓	✓	✓		✓	✓
nPack			✓		✓	✓
FSG		✓	✓			✓
UPX	✓	✓	✓	✓	✓	✓
eXPressor						✓
RLPack	✓					✓
Petite		✓				✓
Aspack		✓	✓			✓
MoleBox		✓	✓		✓	✓
Asprotect		✓				✓
WinUpack	✓	✓	✓	✓	✓	✓
PECompact	✓		✓			✓
Yoda's Crypter						✓
MEW	✓	✓			✓	✓
ORiEN	✓		✓			✓
Private exe Protector		✓	✓			✓
Enigma						✓
ZProtect		✓	✓			✓
Yoda's Protector		✓		✓	✓	✓
Obsidium						✓
SoftwarePassport			✓			✓
Pelock	✓		✓			✓
Telock	✓		✓			✓
Pespin	✓	✓				✓
Armadillo						✓
ACProtect			✓			✓
Themida						

we also study the effectiveness of BareUnpack in analyzing a large set of packed malwares in the wild. Finally, a case study of custom packers is reported.

5.1 Comparative Evaluation with Ground Truth Dataset

Dataset

Our first experiment is to evaluate BareUnpack's effectiveness on famous known packers. We need ground truth dataset so that we can assess experiment results more precisely. For example, we would like to compare BareUnpack's output with the no-packer version to show that BareUnpack perfectly restores the original code. To this end, we use a malware sample hupigon and apply a set of known packers on it. Hupigon family were once in-famous for the back doors they left on the compromised machine. This experiment represents a typical scenario that a malware writer builds a new malware by concealing a malicious code with a third-party packer.

To evaluate BareUnpack successfully extracts the original code, we calculate the code section's MD5 value for both no-packer version and BareUnpack's result.

We also compare BareUnpack's result with other representative generic unpacking tools: PolyUnpack [3], Renovo [4], OmniUnpack [5], CoDisasm [17] and PINdemonium [18]. All of them are relying on some simulated environments. For example, PolyUnpack [3] leverages a debugger, Renovo [4] is based on the emulator of QEMU, OmniUnpack [5] uses the page-level interceptor, while CoDisasm and PINdemonium rely on the dynamic binary instrumentation of Pin [19].

Results

The results of this evaluation are summarized in Table 3. The results show that the comparative generic unpacking tools fail in many cases. We attribute their failures to the simulated environments they used. For example, Armadillo, and Obsidium packers will terminate execution the debugging environment of PolyUnpack [3]; PECompact and Yoda Crypter packers can detect QEMU emulator and circumvent Renovo; PESpin, ACProtect, and Pelock packers can fingerprint Pin environment and crash the execution of CoDisasm and PINdemonium [18].

In contrast, BareUnpack was successful in all cases except Themida. Themida is a sophisticated commercial packer which is based on virtualization obfuscation. BareUnpack extracted some hidden codes which do not match the original binary. We believe that this is the virtualization code equivalent to the original code. Meanwhile, all the comparative generic unpacking tools fail to extract the original codes with Themida.

The evaluation in this subsection illustrates a typical scenario where a malware writer builds a new malware by concealing a malicious code with a packer. The key point here is that BareUnpack effectively retrieves the original code from packed malware and outperform other generic unpacking approaches.

5.2 Analyzing Packed Malware in the Wild

The high effectiveness of BareUnpack enables us to perform large-scale malware analysis. We collect total 71, 259 malware samples from three different malware repositories: VX Heaven[†], VirusShare[††], and VirusTotal[†††]. This malware dataset covers major malware categories such as backdoor, worm, trojan, and virus, including now-infamous ran-

[†]http://vxheaven.org/
[††]http://virusshare.com/
[†††]http://www.virustotal.com/

somware families. The active time of these samples ranges from 2008 to 2017. We report several interesting statistics from our malware dataset: 1) we scan the malware samples with PEiD[†], and 62.4% of them are protected by known packers; 2) only less than 0.2% are only protected by packers with code virtualization; 3) 20.1% of them are packed by custom packers, and this number is increasing over the years; 4) more than 70% of custom packers reveal similar behaviors with known packers (e.g., detecting the run-time environment), which means malware authors customize new packers from existing ones. We will perform a case study on this new type of packers in Sect. 5.3.

Except for the malware protected by code virtualization, we have applied BareUnpack on all of the packed malware successfully. Since we do not have ground truth, we use the "code-to-data" ratio proposed by Eureka [20] to evaluate whether BareUnpack restores the original code. The "code-to-data" ratio for packed code is typically below 5% and above 50% for the unpacked code. Encouragingly, BareUnpack's outputs for all of the packed malware are beyond the threshold of 50% "code-to-data" ratio. Our encouraging results express a strong potential that BareUnpack can be deployed in the practice.

5.3 Case Study: Custom Packers

In our large-scale malware analysis, we find many recent notorious malware families using custom packers to hide their malicious codes. These families are including Conficker, Ogimant, FakeAV, Upatre, Tescrypt, and Cerber Ransomware. The custom packers mean these families are not packed by the known packers written by the third party. Instead, they are packed by the packers written by themselves in order to avoid detection by the AV scanner.

Our in-depth study draws a general finding: most of these custom packers are environment-aware, and they detect the run-time environment to mislead the generic unpacking approaches. Table 4 shows various environment detection methods which used by the malware families' custom packers.

Table 4 The summary of environment detection methods with custom packers of various malware families.

Malware Family	Environment Detection
Conficker	GetTickCount()
Ogimant	GetLastError()
FakeAV	GetCurrorPos()
Upatre	Callback of Graph Controls
Tescrypt	Com Interface

For example, the infamous backdoor of Conficker uses the API of "GetTickCount" to detect the simulated environment, as many simulated environments can not simulate this API just like the real OS [21], [22]. And the recent Cerber Ransomware will crash the sandbox environment while run well in the bare-metal OS.

We also evaluate these environment-aware custom packers with BareUnpack and other comparative unpacking approaches. The results are shown in Table 5.

The results show that BareUnpack outperforms the comparative unpacking approaches when applied to the environment-aware custom packers. We also attribute the failures of comparative unpacking approaches to their simulated environments. For example, the QEMU emulator of Renovo can not emulate the API "GetTickCount" correctly. And both CoDisasm and PINdemonium can not implement the API of GetLastError like the bare-metal OS.

The evaluation in this subsection illustrates another typical scenario where unpack the prevalent malware protected by the custom packer. As more and more prevalent malware families use custom packers in recent years [2], we believe that BareUnpack can improve the efficiency of security researchers when analyzing these prevalent malwares.

6. Discussion

Since BareUnpack works with adversaries, we have to consider how a skilled attacker could circumvent BareUnpack once our approach is known. In this section, we discuss BareUnpack's possible attacks and limitations, which also light up our future work.

6.1 Possible Attacks

Recall that BareUnpack attempts to work in bare-metal OS, without the use of any detectable component. The only modification of to the OS by BareUnpack is the "GetProcAddress" hooking. Therefore, we have to consider how a skilled attacker could circumvent BareUnpack by detect the "GetProcAddress" hooking.

A method for preventing hooking detection from memory scanning is commonly known as the memory subversion technique [23]. Memory subversion was first proposed in the Shadow Walker rootkit [24] to prevent detection from memory scanning. The Shadow Walker rootkit demonstrated that it was possible to control the view of memory

Table 5 Comparative evaluation of unpacking capability with custom packers.

Packers	PolyUnpack	Renovo	OmniUnpack	CoDisasm	PINdemonium	BareUnpack
Conficker	✓		✓	✓	✓	✓
Ogimant	✓	✓	✓			✓
FakeAV						✓
Upatre	✓		✓	✓	✓	✓
Tescrypt	✓		✓	✓	✓	✓
Cerber ransomware						✓

[†]https://www.aldeid.com/wiki/PEiD

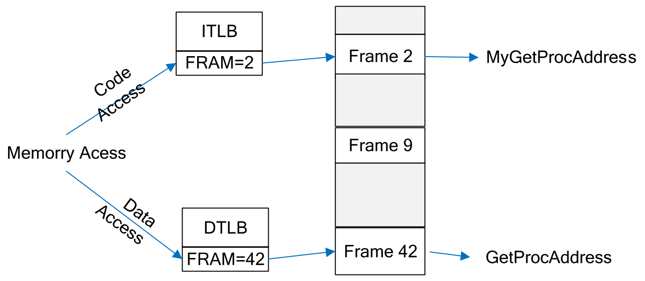

Fig. 8 The memory-subversion.

regions seen by OS and other processes via exploiting the Intel split TLB (Translation Lookaside Buffer) architecture. The basic idea of memory subversion is desynchronizing ITLB and DTLB translate code and data access Inspired from Shadow Walker rootkit, we improve BareUnpack, using memory subversion mechanism to forward the code access of the logic frame (ITLB)to the physical frame which contains "MyGetProcAddress" and translate the data access (DTLB)to "GetProcAddress" (shown as Fig. 8). In this way, BareUnpack can bypass the detection from memory scanning.

6.2 Limitation

Virtualization packers, such as Themida, pose a extreme case problem for all the unpackers. These packers do not reveal the original payload at run-time. Instead, the original payload is replaced by several bytecode, and the attached virtualization engine will simulate these bytecode at run time. irtualization packers represent a completely different challenge, we plan to how to handle this challenge in the future work.

7. Related Work

Hidden code extraction from the packed executables, which called unpacking, is the major challenge to malware analysis. Generic unpacking is a major solution to the threat of various packing techniques. Emulator, debugger, and virtual machine, which used as the simulated environments for the existing generic unpacking approaches to unpack the packed executables [15].

PolyUnpack [3], Renovo [4], OmniUnpack [5] CoDisasm [17] and PINdemonium [18] are the most notably generic unpacking approaches. Table 6 illustrates the various simulated environments applied by these approaches.

PolyUnpack [3] first builds a static view of the program, and then uses a debugger to single step the execution of the program to check the executed code whether outside the static view.

Renovo [4] uses the emulator of QEMU to track execution of the program, identifying the newly generated code as the hidden code.

OmniUnpack [5] leverages the page-level interceptor to monitor the execution of the program. It identifies the code executing in a page which was newly modified as the hidden code.

Table 6 The summary of the existing generic unpacking approaches.

Approach	Simulated Environment	Monitoring Granularity	Evade Evasions
PolyUnpack	Debugger	Instruction	Anti-Debugger
Renovo	Emulator	Instruction	Anti-Emulator
OmniUnpack	VM	Page	Anti-VM
CoDisasm	Pin	Instruction	Anti-Pin
PinDemonium	Pin	Instruction	Anti-Pin

CoDisasm [17] developed Pin tracer to collects an execution trace of a stripped binary, which is based on Pin tool [19] and recovers each unpacking layer by taking a memory snapshot at the beginning of the layer. Similarly, PinDemonium [18] also relies on the Pin tool to "identify written and then executed memory regions".

These simulated environments have many obvious differences with the bare-metal OS. The environment sensitive packer can be aware of the existence of the simulated environment and then change its behavior to evade unpack.

The limitation of simulated environments motivates us to design an transparent unpacking approach on the bare-metal OS directly, not relying any simulated environment. Our solution ensures that we can unpack the environment-aware packers effectively.

8. Conclusion

To evade detection, malware writer often tries to pack the original executables into packed one.

The existing generic unpacking approaches need the simulated environments to monitor the execution of the packed executables. The simulated environments make these approaches easily evaded by the environment-sensitive packers. In this paper, we propose a new execution monitoring method, called "packed IAT hooking", which can monitor the execution of packed executables on the bare-metal OS. And then we propose BareUnpack, a bare-metal generic unpacking approach which is based on the "packed IAT hooking". BareUnpack can unpack the packed executables on the bare-metal OS directly, not requiring any simulated environment.

Our experimental evaluation has shown that BareUnpack successfully unpacks majority of packers and that performs well compared to existing generic unpackers. We believe that BareUnpack can help the security researchers to cope with the growing threat of packed malwares.

Since the GOT table of ELF file in Linux OS has the similar function to the IAT of PE file in Windows OS. Both of them list functions to import from other shared libraries. We plan to study the feasibility of applying BareUnpack's idea to packed Linux malware in the future.

Acknowledgments

This research was supported by the Natural Science Foundation of Hubei Province of China (2017CFB307, 2018CFB709), Talent Introduction Project of Hubei Normal University in 2017, College of Arts and Science of Hubei

Normal University Research project (Ky201803), CERNET Innovation Project (NGII20160411), and innovative team of excellent young-middle-aged, universities in Hubei province (T201430).

References

[1] F. Guo, P. Ferrie, and T.C. Chiueh, "A study of the packer problem and its solutions," Proc. 11th International Symposium on Recent Advances in Intrusion Detection (RAID'08), 2008.

[2] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P.G. Bringas, "Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers," 2015 IEEE Symposium on Security and Privacy (SP), pp.659–673, IEEE, 2015.

[3] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "PolyUnpack: Automating the hidden-code extraction of unpack-executing malware," Proc. 22nd Annual Computer Security Applications Conference (ACSAC'06), 2006.

[4] M.G. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code extractor for packed executables," Proc. 5th ACM Workshop on Recurring Malcode (WORM'07), pp.46–53, Nov. 2007.

[5] L. Martignoni, M. Christodorescu, and S. Jha, "OmniUnpack: Fast, generic, and safe unpacking of malware," Proc. 23nd Annual Computer Security Applications Conference (ACSAC'07), 2007.

[6] D. Kirat and G. Vigna, "Malgene: Automatic extraction of malware analysis evasion signature," Proc. 22nd ACM SIGSAC Conference on Computer and Communications Security, pp.769–780, ACM, 2015.

[7] D. Kirat, G. Vigna, and C. Kruegel, "Barecloud: Bare-metal analysis-based evasive malware detection," USENIX Security Symposium, pp.287–301, 2014.

[8] C. Wressnegger, K. Freeman, F. Yamaguchi, and K. Rieck, "Automatically inferring malware signatures for anti-virus assisted attacks," Proc. 2017 ACM on Asia Conference on Computer and Communications Security, pp.587–598, ACM, 2017.

[9] L. Sun, T. Ebringer, and S. Boztas, "An automatic anti-anti-vmware technique applicable for multi-stage packed malware," 2008 3rd International Conference on Malicious and Unwanted Software (MALWARE), pp.17–23, IEEE, 2008.

[10] M. Lindorfer, C. Kolbitsch, and P.M. Comparetti, "Detecting environment-sensitive malware," Recent Advances in Intrusion Detection, pp.338–357, Springer, 2011.

[11] H. Father, "Hooking windows api-technics of hooking api functions on windows," CodeBreakers J., vol.1, no.2, 2004.

[12] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox," IEEE Security & Privacy, vol.5, no.2, pp.32–39, 2007.

[13] H.c. WANG, Y. SHI, and Z. XUE, "Research and implementation of secure password input under windows," Information Security and Communications Privacy, vol.4, pp.53–55, 2011.

[14] J. Berdajs and Z. Bosnić, "Extending applications using an advanced approach to dll injection and api hooking," Software: Practice and Experience, vol.40, no.7, pp.567–584, June 2010.

[15] K.A. Roundy and B.P. Miller, "Binary-code obfuscations in prevalent packer tools," ACM Comput. Surv. (CSUR), vol.46, no.1, p.4, Oct. 2013.

[16] D. Korczynski, "Repeconstruct: reconstructing binaries with self-modifying code and import address table destruction," 2016 11th International Conference on Malicious and Unwanted Software (MALWARE), pp.1–8, IEEE, 2016.

[17] G. Bonfante, J. Fernandez, J.Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry, "Codisasm: medium scale concatic disassembly of self-modifying binaries with overlapping instructions," Proc. 22nd ACM SIGSAC Conference on Computer and Communications Security, pp.745–756, ACM, 2015.

[18] S. D'Alessio and S. Mariani, "Pindemonium: a dbi-based generic unpacker for windows executables," Proc. 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'16), 2016.

[19] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," Acm sigplan notices, pp.190–200, 2005.

[20] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee, "Eureka: A framework for enabling static malware analysis," Proc. 13th European Symposium on Research in Computer Security (ESORICS'08), vol.5283, pp.481–500, 2008.

[21] S. Shin and G. Gu, "Conficker and beyond: a large-scale empirical study," Proc. 26th Annual Computer Security Applications Conference, pp.151–160, ACM, 2010.

[22] S. Shin, G. Gu, N. Reddy, and C.P. Lee, "A large-scale empirical study of conficker," IEEE Trans. Inf. Forensics Security, vol.7, no.2, pp.676–690, April 2012.

[23] G.L. Garcia, "Forensic physical memory analysis: An overview of tools and techniques," TKK T-110.5290 Seminar on Network Security, pp.305–320, 2007.

[24] S. Sparks and J. Butler, "Shadow Walker: Raising the bar for windows rootkit detection," Black Hat Japan, vol.11, no.63, pp.504–533, 2005.

Binlin Cheng received his Ph.D. degree from Wuhan University in 2016, China. Now he is a lecturer in Hubei Normal University. His main research interests include software security and network security.

Pengwei Li received his Ph.D. degree from Wuhan University in 2017, China. Now he is a lecturer in Nanjing Audit University. His main research interests include software security and network security.