Reconstructing a Packed DLL Binary for Static Analysis*

Xianggen Wang[1,2], Dengguo Feng[2], and Purui Su[2]

[1] University of Science and Technology of China
[2] State Key Laboratory of Information Security, Institute of Software, Chinese Academy of Sciences

Abstract. DLLs (Dynamic Link Libraries) are usually protected by various anti-reversing engineering techniques. One technique commonly used is code packing as packed DLLs hinder static code analysis such as disassembly. In this paper, we propose a technique to reconstruct a binary file for static analysis by loading a DLL and triggering and monitoring the execution of the entry-point function and exported functions of packed DLLs. By monitoring all memory operations and control transfer instructions, our approach extracts the original hidden code which is written into the memory at run-time and constructs a binary based on the original DLL, the codes extracted and the records of control transfers. To demonstrate its effectiveness, we implemented our prototype ReconPD based on QEMU. The experiments show that ReconPD is able to analyze the packed DLLs, yet practical in terms of performance. Moreover, the reconstructed binary files can be successfully analyzed by static analysis tools, such as IDA Pro.

Keywords: Security Analysis, Malware Analysis, Binary Reconstructing, Dynamic Analysis, Static Analysis.

1 Introduction

Reverse engineering becomes a prevalent technique to analyze malware. To make the analysis more difficult, malware writers usually protect their programs including executables and DLLs by various anti-reverse engineering techniques. One of the most popular anti-reverse engineering methods is binary code packing [18,20]. It is impossible to do static analysis on packed malware because of the inaccuracy of disassembler. This paper focuses on identifying and extracting the hidden code generated using binary code packing and reconstructing a binary for static analysis.

To identify and extract the hidden code in packed binary programs various tools have been developed. However, most of commonly known tools such as PEiD [4]

* Supported by National Natural Science Foundation of China (No.60703076) and National High Technology Research and Development Program of China (No.2006AA01Z412 and No.2007AA01Z451)

F. Bao, H. Li, and G. Wang (Eds.): ISPEC 2009, LNCS 5451, pp. 71–82, 2009.

are only valid for some special known packing algorithms applied on the packed executables; what is more, they are usually unable to analyze packed DLLs.

DLL is commonly used in the malware. Windows makes certain features available only to DLLs. For example, you can install certain hooks (set using **SetWindowsHookEx** and **SetWinEventHook**) only if the hook notification function is contained in a DLL. As a number of critical technologies and codes are packed in DLL, it is becoming one important target of anti-reverse engineering protection.

By now, the difficulty of statically analyzing a packed binary program is how to correctly disassemble them. One of approaches to solve this problem is to reversely analyze the packing algorithm and develop specific automatic unpacking tools. But this method is only valid for known packing tools(such as UPX). In fact, it is easy for malware writers to implement new anti-reverse engineering algorithms and tricks. Dynamic analysis is a promising solution to the problem of hidden code extraction because it does not depend on signatures. The original code or its equivalent must eventually be restored in memory and get executed at some point at run-time regardless of what packing algorithms might be applied [20]. By taking advantage of this intrinsic nature of packed binary programs, we could potentially extract the original hidden binary code or its equivalent and reconstruct a binary for static analysis by patching the extracted code on the original packed binary program. Most of the work to date on identification and extraction of the original hidden code has focused on executables. This paper, by contrast, focuses on the DLL.

In this paper, we present a fully dynamic approach for automatically identifying packed DLLs, and extracting the original hidden code which is dynamically generated and executed at run-time and additional information useful for reconstructing of the packed binary. Finally we reconstruct a binary which can be directly analyzed by static analysis tools, such as IDA Pro. This method is valid for unknown packing algorithms without the complex reverse analysis. In summary, the contributions of this paper are as follows:

1. **Propose a general and fully dynamic approach for identifying and extracting the original hidden code of packed DLLs:** Previous work relies on either heuristics of known packing tools or the accuracy of the disassembler and is only valid for packed executables. In this paper, we present a binary extraction technique which is applicable to packed DLLs. This method is fully dynamic and thus does not depend on the program disassembly or the known signatures of packing algorithms. It triggers the execution of entry-point function and exported functions of a DLL, monitors the execution and records related memory operations by shadow memory and extracts the content of the memory which is dynamically generated and executed.
2. **Provide a method to reconstruct a DLL binary file based on the execution trace:** By patching the extracted code on an original packed DLL binary file, we reconstruct a binary file which can be directly disassembled.
3. **Implement and evaluate ReconPD, an automated framework for extracting hidden code and reconstructing binary files.** Applying

our proposed technique, we build a framework for automatically examining DLL binaries, extracting their original hidden code and reconstructing a binary file based on the extracted code and additional information. The reconstructed binary file can be successfully analyzed by static techniques. Based on the prototype, we have successfully done a series of experiments on the analyses of packed DLL binary files. We also present the evaluation results of ReconPD, demonstrating that it is applicable to analyze packed DLL binary.

The rest of the paper is organized as follows: In Section 2 we review the related work. Section 3 presents the problem definition and describes the proposed method. In Section 4 we provide details of the technique and the prototype implementation. Section 5 shows the experimental results. Finally, Section 6 concludes the paper.

2 Related Work

Static analysis is one of the most popular software analysis methods nowadays, it is widely used in software reverse engineering [1] and software security analysis [2,3]. Static analysis tools can analyze intermediate binary code that is unable to run, since they don't need to really execute the code at all. And they have a good global view sight of the whole binary code and can figure out the entire program logic before running it.

To protect the privacy of software and prevent reverse engineering, software writers usually have their programs heavy-armored with various anti-reverse engineering techniques [7,8]. Such techniques include SMC (Self-Modifying Code) [9,10], encryption [11,12], binary and source code obfuscation [14,15], control-flow obfuscation [13], instruction virtualization [5,6], and binary code packing [18]. The development of anti-reverse technology makes static analysis more and more difficult. Although dynamic analysis can compensate static analysis for its effectiveness and accuracy to a certain extent, static analysis has some advantages that dynamic analysis doesn't have such as the more comprehensive view of a program's behavior.

Currently, extracting and re-building the original program from a protected binary has been one of the major challenges for software reverse engineers and security community.

Identifying and extracting the original hidden code
PolyUnpack [18] is a general approach for extracting the original hidden code without any heuristic assumptions. As a pre-analysis step, PolyUnpack disassembles the packed executable to partition it into the code and data sections, and then it executes the binary instruction by instruction, checking whether the instructions are in the code section or not. However, in terms of performance, disassembling binary executables as being done in [18] and [19] is an arduous task and correctly disassembling a binary program is challenging and error-prone, as demonstrated in [19]. In Addition, PolyUnpack can not extract the hidden code inside the DLL.

Min Gyung Kang et al. proposed a fully dynamic method Renovo [20] which monitors currently-executed instructions and memory writes at run-time. This approach inserts a kernel module into the emulated system and registers several call-back functions, by observing the program execution. Compared with it, our approach doesn't need touch anything in GusetOS and is completely transparent to analysis target; moreover, our method is applicable to a packed DLL binary and we can reconstruct a binary file for static analysis by monitoring the control transfers.

Re-building the original program
PEiD [4] is a tool for identifying compressed Windows PE binaries. Using the database of the signatures for known compression and encryption techniques, it identifies the packing method employed and suggests which unpacker tool can be applied. However, despite their ability to perfectly restore the original program, executables packed with unknown or modified methods are beyond the scope of this approach.

Christodorescu et al. proposed several normalization techniques that transform obfuscated malware into a normalized form to help malware analysis [16]. Their unpacking normalization is similar to our approach. Its basic idea is to detect the execution of newly-generated code by monitoring memory writes after the program starts. Our approach goes further; we identify and monitor memory modification and execution in all code, data, stack and heaps, and also our approach can tackle dynamic code generation inside the packed DLL binary.

3 Problem Statement

One of the most popular anti-reverse engineering methods is binary code packing which is usually used to protect binary program against reverse engineering and other attacks. Code packing transforms a program into a packed program by encrypting or compressing the original code and data into packed data and associating it with a restoration routine [20]. It is impossible to analyze the packed binary program using static techniques, thus, the target of our work is to identify them and extract the original hidden code in packed binary files. We generate a binary file which can be successfully analyzed by static techniques by patching a packed binary file with extracted code and restoring its control transfers.

The protecting procedure can be expressed as below:

$$dummy_i = HR_i(target_i, key_i)$$

A *hiding routine* HR_i is a set of instructions which camouflage a target instruction $target_i$ with a dummy instruction $dummy_i$ based on encrypt key key_i. The key may be null or an invariant if the hiding routine is just a transformer.

A *restoring routine* RR_i is a set of instructions which un-camouflage an original target instruction hidden by a dummy instruction based on the decrypt key key_i. RR_i can be viewed as a set of instructions that replaces $dummy_i$ with $target_i$ at run-time [9]. Fig.1 shows how a packed program works [20].

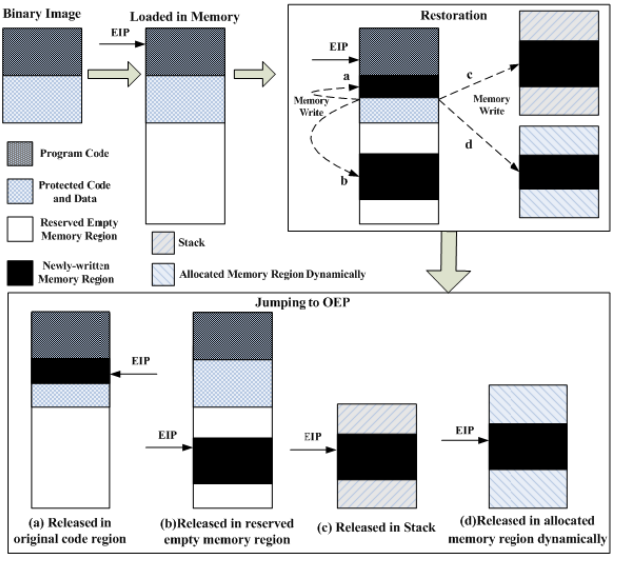

Fig. 1. How a packed program works

The restoring procedure during execution can be described as below:

$$target_i = RR_i(dummy_i, key_i)$$

RR_i is a reverse procedure of HR_i. In an analysis, HR_i is unavailable usually, so we can only analyze RR_i. But it's difficult and time-consuming to analyze RR_i directly, so we observe the execution of RR_i in order to identify and extract the hidden code it releases.

In the following discussion, we assume there is a valid key, and also we assume that at least a part of dynamically released code will be executed at run-time.

RR_i may be only used to release the code, or used to release both code and data. In the procedure of reconstructing a binary, we must disable the RR_i code generation but retain its data generation. Otherwise, it may disturb the reconstructed binary's execution.

In the procedure of restoring code, the memory used to store code may be used to store one block of code or blocks of code iteratively. That is, there may be several $dummy_i$. We suppose that one of them is $dummy_i$ whose address is A_d^i, and the related code released is $target_i$, whose address is A_t^i. There are two different models:

1. for any $i, j > 0$, and $A_d^i \neq A_d^j$, there is $A_t^i \neq A_t^j$;
2. there exist $0 < i < j$, making $A_t^i = A_t^j$,but $A_d^i \neq A_d^j$;

In the first model, each restored code block is stored in different memory regions, that is, they do not overlap with each other. In the second, there are at least two released blocks stored in the same memory region, resulting in some newly-generated code overlapping with each other. For the second model, we need to

transform the addresses because of the address confliction when we reconstruct the binary. The details will be discussed in Section 4.

4 System Design and Implementation

To demonstrate the effectiveness of our approach, we design and implement a system, ReconPD, to automatically identify packed DLLs and extract their original hidden code. Fig.2 depicts an overview of ReconPD. The prototype ReconPD is a subsystem of WooKoo, a malware dynamic binary analysis platform, which was developed by us based on QEMU [22]. We modified QEMU to monitor the dynamic execution of a target executed in the Guest OS.

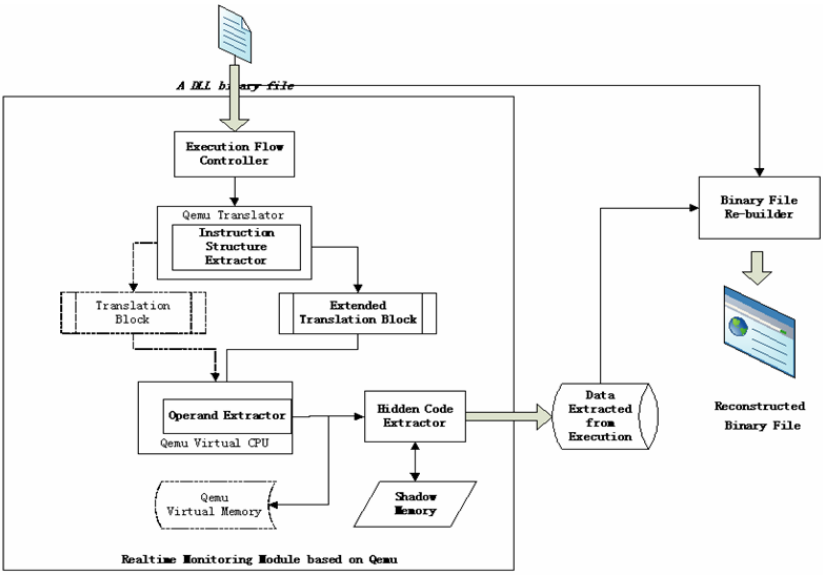

Fig. 2. The Framework of ReconPD

ReconPD extended QEMU Translator by adding the module of Instruction Structure Extractor and extended QEMU Virtual CPU by adding the module of Operand Extractor. It also extended the data structure of TB (Translation Block) to store the information of the instruction structure and added a new data structure of shadow memory. When analyzing a DLL, we run it in an *emulated environment*. *Execution Flow Controller* loads a DLL and triggers the entry-point function and exported functions. *Hidden Code Extractor* observes its execution, consults the *shadow memory*, and determines if any hidden code is currently executed. If so, it extracts the hidden code and other data. Finally, *Binary File Re-builder* reconstructs a binary file based on the original DLL and the data collected by Hidden Code Extractor. We will present these components

and provide some implementation details in turn, and discuss the limitations of the current implementation of ReconPD.

4.1 Recognizing and Extracting the Hidden Code

No matter what packing methods are applied and where the hidden code is restored, the original program code and data should eventually be present in memory to be executed, and also the instruction pointer should jump to the OEP of the restored program code which has been written in memory at runtime [20]. Taking advantage of this inevitable nature of packed DLLs, we propose a technique to dynamically extract the hidden original code from the packed DLLs by examining whether the current instruction is newly-generated after the DLL is loaded.

Execution Flow Controller. Before an application (or another DLL) can call functions in a DLL, the DLL's file image must be mapped into the calling process' address space. Normally, when a DLL is mapped into a process' address space, the system calls a special function in the DLL (we call it the entry-point function, usually **DllMain**) which is used to initialize the DLL. Usually the restoration of the hidden code and data is a part of the initialization process for packed DLL.

Windows is unable to launch DLL directly, so in this paper we make use of a DLL loader named LoadDLL.exe for loading a DLL by calling Windows API **LoadLibraryEx**. In order to monitor the execution of the entry-point function of a DLL, we set *dwFlags* to **DONT_RESOLVE_DLL_REFERENCES** to prevent the operation system from calling the entry-point function to initialize the DLL. To obtain a comprehensive analysis of the DLL as the entry-point function is executed, we control the execution flow and trigger all exported functions of the DLL. As the return instruction of the entry-point function is "RET 0C", we take it as a sign of the end of the execution of the entry-point function. An exported function is typically allowed to run until it exits normally or crashes.

Considering that determining whether a program has hidden code or not is an *un-decidable* problem [18], we introduce a configurable time-out parameter into the system. We terminate the extraction procedure if we do not observe any hidden code being executed within this time-out. In the experiments, we set this parameter to be 30 minutes.

Shadow Memory. Based on the above inevitable instinct nature of packed DLLs, shadow memory is introduced to record every memory modification. According to whether the written content is executed or not, we can determinate whether it is code or data. Thus, we need to record the following information for each byte of memory:

1. EIP_M: The instruction modifying this byte.

2. EIP_C: The instruction referring this byte, such as directly jumping to this memory by JMP and JNZ, or calling the function whose entry point is this memory.
3. *State*: The state of this byte, there are three possible states, *clean*, *dirty*, and *executed*. Fig.3 shows how the states transfer.

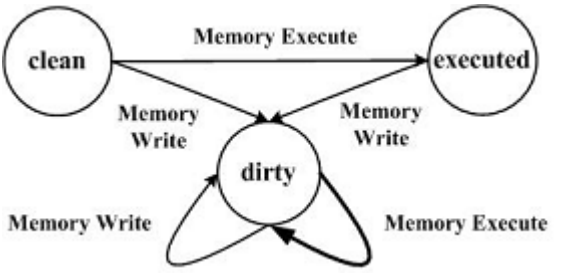

Fig. 3. How a memory state transfers

We just need to pay attention to the state *dirty* and *clean* in order to recognize and extract the original hidden code. In ReconPD, we assign a block of shadow memory for each monitored memory region (including static region and dynamic region such as stack and heaps allocated during execution) of the DLL-calling process (LoadDll.exe).

In order to reconstruct the control flow, we record *control transfer instructions*, such as JNZ and CALL, which will change the execution flow. During the reconstruction of a binary file, we modify their destination address if the target code is reassigned.

Single-Step Monitoring. Our monitoring on the execution needs to intercept each instruction to get the real values of operands, so we make the monitored code execute in single-step mode and each TB contain one instruction in QEMU.

In QEMU, there is a flag of virtual CPU for single-step execution mode. If the flag is set, the code of both operation system and applications is executed in single-step mode. Furthermore, the flag is perceptible for applications in the GuestOS, and the analyzed object may detect it and destroy itself or stay inactive if it found the flag is set. In ReconPD, an additional flag, $TSingleStep$, was introduced, which is transparent for applications. When $TSingleStep$ is set, the translator makes sure that there is only one instruction in each TB of the target process, and instructions of other process are executed as normal.

By the method above, we not only keep the analysis transparent for the target process but also improve the analysis efficiency by avoiding the whole system running in single-step mode.

Instruction Interception. During monitoring the execution, we intercept the following two classes of instructions:

Memory Modifications: including MOV, MOVS, STOSB and so on. These operations are used to write dynamic code into memory. The stack operations such as POP are not included, which are rarely used to generate code.

Control Transfers: including direct jumps/calls, indirect jumps/calls, and return instructions, which determine the control flow of the execution.

ReconPD firstly records the structure of instruction during the translation and extracts the real values of operands during run-time.

4.2 Reconstructing Binary Files

In this phrase, we patch the hidden code exacted on the packed DLL binary file and restore the control transfers to generate a binary file for static analysis. There are two parts of work during this phrase.

Patch the original hidden code extracted on the packed DLL binary file

1. The code is released in code region and covers the original protected code just as showed in Fig2(a). We write the dynamic code into the binary file where this block is mapped from the binary file.
2. The code is released in reserved empty memory of code region without covering the original code just as showed in Fig2(b), we set $SizeOfRawData$ of the code section to $VirtualSize$, then we write the dynamic code into the revised binary file where this block is mapped from the binary file.
3. The code is released in data region, such as stack and heap just as showed in Fig2(c,d), we expand a new section into the PE binary and the code is written into the expanded section. It's a mature technique to expand the PE binaries, thus, we do not discuss it in this paper.
4. There are two or blocks written into the same memory being covered each other, the first block write into the binary file where this block is mapped from, and the rest are written into the expanded section in the binary file.

In the restoration process, we should modify *dest* of the control transfer to the correct address where the instruction is reassigned in a binary file. New value of *dest* can be easily got by address mapping. But sometimes, we can not directly update it. For example, some branch instructions like "jmp ebx;" may transfer control to different dynamic generated code blocks at different conditions. For this case, we make the branch instruction transfer control to a dispatch routine, which will mimic the original branch conditions to call different dynamic generated code blocks at different time. By this way, though we can not completely rebuild the original logic of the execution, we can still provide all possible control transfers, which is much helpful for static analysis.

Disable the restoration, preventing it from modifying the patched code. When releaser's code generation behavior is disabled, we should consider whether the releaser dynamically modifies code only or both code and data. If a releaser only modifies code, we could disable it simply by replacing it with NOP

instruction. Or else, to keep the original program logic, we have to reserve its function of modifying data and prevent it from modifying code. So we replace the releaser with a unconditional jump, which transfers control to a judge function in the new code section, and the judge function is an encapsulation of the releaser, which behaves just like the releaser if the modification's target is data but does nothing if the target is code.

5 Evaluation

The experiments are finished on the WooKoo Platform, based on Fedora Core 3 Linux 2.6.9-1.667smp and QEMU 0.8.2. The experiments are all finished with GuestOS of Windows XP Professional SP2. In this section, we describe two experiments and present the evaluation results, demonstrating that ReconPD is an accurate and practical solution for extracting the original hidden code of packed DLLs and the reconstructed binaries can be successfully analyzed by static analysis tools such as IDA pro.

5.1 Synthetic Samples

Because of the difficulty of collecting wild samples, we have to develop some samples. To verify that ReconPD generates accurate results, we have tested ReconPD against the synthetic sample programs generated by using a packing tool developed by us. The packing tool applies different packing techniques as well as encryption, code obfuscation to thwart reverse engineering.

We use some DLLs belong to Microsoft Windows Operating System as the original binary to generate synthetic packed program samples. With the knowledge that the packing tool usually restore and execute the original binary instructions at run-time, we could verify the correctness of our extraction technique by comparing the extracted hidden code regions with the .text section of the original binary.

Table 1. The Experimental Results of Packed DLLs

Files	Size of Packed Files(KB)	Size of Shadow Memory(KB)	Time without monitor(Seconds)	Time under monitor (Seconds)	Reconstructed Binary
rasapi32.dll	234	2928	54	62	Loadable
jscript.dll	444	5376	38	45	Loadable
mfc42.dll	1008	12240	32	39	Loadable
msi.dll	2792	33744	22	25	Loadable
shell32.dll	8112	97728	674	681	Loadable

As shown in Table 1, ReconPD fully extracted the original binaries processed by our packing tools. And the reconstructed DLL binary files could be loaded and disassembled by IDA Pro properly. From the experimental results, we found the size of shadow memory is almost 12 times of DLLs.

5.2 Performance Overhead

We evaluated the performance by running the samples with and without monitor in single step mode. From the results shown in Table 1, we found that the monitor decreased the performance about 18.9%. Without the monitor, we also compared the performance of running in single step mode with that of running in normal mode. We found that the performance decreased greatly by single step mode even we have optimized it by only keeping the target process in single-step mode but not the whole system as QEMU. Considering that ReconPD is aiming to provide hidden code extraction environment for malware analysis which usually takes several hours to days, this degree of slowdown in initial execution time is tolerable.

5.3 Limitations

Just as discussed above, there are some limitations of ReconPD.

Some exported functions crash at run-time because of the improper parameters provided by us. We are able to analyze the number of parameters of exported functions but not recognize the data type of parameters. We may patch both the data and code to reconstruct complete binary while there may be no comprehensive solution to this problem. We have intercepted typical memory write instructions. However, there is a method to evade the monitor by replacing some instructions. For example, *"XOR EAX,EAX; ADD EAX,EBX"* can replace the instruction *"MOV EAX,EBX;"*. In the future, we will make it support much more instructions.

Finally, some of the reconstructed binary are unable to be loaded because ReconPD does not patch the data. It just monitors and restores both the original hidden code and control transfers.

6 Conclusion

We proposed a technique to reconstruct a binary file for static analysis by monitoring the executions of packed DLLs. This method is valid for unknown protection algorithms without the complex reverse analysis. To demonstrate the idea, we have implemented ReconPD based on QEMU. By triggering the execution of entry-point function and exported functions of a DLL, and monitoring all the memory operations and control transfer instructions, it identifies and extracts the code which is written into the memory during execution and constructs a binary file based on the original binary, the codes extracted and the records of control transfers. We show that ReconPD can successfully analyze packed DLLs and the reconstructed binary can be successfully analyzed by static analysis tools, such as IDA Pro.

References

1. Balakrishnan, G., Gruian, R., Reps, T., Teitelbaum, T.: CodeSurfer/x86—A platform for analyzing x86 executables. In: Bodik, R. (ed.) CC 2005. LNCS, vol. 3443, pp. 250–254. Springer, Heidelberg (2005)

2. Christodorescu, M., Jha, S.: Static Analysis of Executables to Detect Malicious Patterns. In: Usenix Security Symposium (2003)
3. Christodorescu, M., Jha, S., Seshia, S., Song, D., Bryant, R.: Semantics-aware Malware Detection. In: IEEE Symposium on Security and Privacy (2005)
4. PEiD, http://www.secretashell.com/codomain/peid/
5. Themida, http://www.oreans.com/
6. Yoda Protector, http://sourceforge.net/projects/yodap/
7. van Oorschot, P.C.: Revisiting software protection. In: Boyd, C., Mao, W. (eds.) ISC 2003. LNCS, vol. 2851, pp. 1–13. Springer, Heidelberg (2003)
8. Wang, P.: Tamper Resistance for Software Protection, Master Thesis, Information and Communications University, Korea (2005)
9. Kanzaki, Y., Monden, A., Nakamura, M., Matsumoto, K.: Exploiting self-modification mechanism for program protection. In: Proc. of the 27th Annual International Computer Software and Applications Conference, pp. 170–181 (2003)
10. Giffin, J.T., Christodorescu, M., Kruger, L.: Strengthening Software Self-Checksumming via Self-Modifying Code. In: 21st Annual Computer Security Applications Conference, pp. 23–32 (2005)
11. Albert, D.J., Morse, S.P.: Combating Software Piracy by Encryption and Key Management. Computer (1984)
12. Lee, J.-W., Kim, H., Yoon, H.: Tamper resistant software by integrity-based encryption. In: Liew, K.-M., Shen, H., See, S., Cai, W. (eds.) PDCAT 2004. LNCS, vol. 3320, pp. 608–612. Springer, Heidelberg (2004)
13. Huang, Y.L., Ho, F.S., Tsai, H.Y., Kao, H.M.: A control flow obfuscation method to discourage malicious tampering of software codes. In: ASIACCS 2006, computer and communications security, New York, NY, USA, p. 362 (2006)
14. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: CCS 2003, New York, NY, USA, pp. 290–299 (2003)
15. Wroblewski, G.: General method of program code obfuscation. In: Proc. Int. Conf. on Software Engineering Research and Practice (SERP) (2002)
16. Christodorescu, M., Kinder, J., Jha, S., Katzenbeisser, S., Veith, H.: Malware normalization. Technical Report 1539, University of Wisconsin, USA (2005)
17. DataRescue SA. IDA Pro disassembler: Multi-processor, Windows hosted disassembler and debugger, http://www.datarescue.com/idabase/
18. Royal, P., Halpin, M., Dagon, D., Edmonds, R., Lee, W.: PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In: ACSAC 2006, USA, pp. 289–300 (2006)
19. Nanda, S., Li, W., Lam, L., Chiueh, T.: BIRD: Binary interpretation using runtime disassembly. In: CGO 2006, USA, pp. 358–370 (2006)
20. Kang, M.G., Poosankam, P., Yin, H.: Renovo: A Hidden Code Extractor for Packed Executables. In: The 5th ACM Workshop on Recurring Malcode (WORM) (2007)
21. Moser, A., Kruegel, C., Kirda, E.: Exploring Multiple Execution Paths for Malware Analysis. In: SP 2007, pp. 231–245 (2007)
22. Bellard, F.: QEMU, a Fast and Portable Dynamic Translator. In: FREENIX Track: 2005 USENIX Annual Technical Conference (2005)