Toward Generic Unpacking Techniques for Malware Analysis with Quantification of Code Revelation

Hyung Chan Kim, Daisuke Inoue, Masashi Eto, Yaichiro Takagi, Koji Nakao

Information Security Research Center
National Institute of Information and Communications Technology
Nukui-kitamachi 4-2-1, Koganei 184-8795, Japan
{hckim, dai, eto, yaichiro, ko-nakao}@nict.go.jp

Abstract. Many malwares we encounter these days are armed with its own custom encoding methods (i.e., they are packed) to make security/reverse analysis difficult or impossible. Dealing with large volumes of malware samples, building general unpacking framework is important to lift the burden of manual post processing. In this paper, we present our preliminary efforts toward generic binary unpacking. Our architecture includes a concept of shadow states to observe the encoding/decoding activities of stub code in a packed malware and quantify the revealed code resulted from those activities to spot candidate original entry points automatically. We show our proof-of-concept implementation of the proposed method with dynamic binary instrumentation technique. We confirmed the effectiveness of our method with the experimental results conducted on some widely deployed packers and also identified its limitation.

Keywords: generic unpacking, malware analysis, software security, operating systems, dynamic binary instrumentation

1 Introduction

In these days, honeypot systems, operated in malware analysis groups or incident response teams, encounter with numerous malware samples day by day and the backlog for post handling of those samples are getting longer. Unfortunately, large portions of such malware samples seem to be packed (i.e., compressed, encrypted, and/or obfuscated) to lurk themselves thereby making analysis of real malware substances difficult or even impossible. Applying packing in malwares also hinders detectability of anti-virus software even though malwares under analysis are already known and their signatures are available [1] due to that a malware packed with various packers might gain polymorphism to a certain degree (or with a single packer that supports polymorphism itself).

Our research group has a collection of malwares size of 643409 (as of May, 2009). Among them, at around 50% of samples are identified to be packed with PEiD tool [2] with a community supported signature database [23]. As signature based identification often shows false negative results, we also conducted with an entropy

identification [3] scheme, naively setting thresholds (6.7 for file and 7.2 for highest section entropy), and it resulted in that almost 90% of samples outnumbered the thresholds. This tendency comes from the publicly available packing tools, sometimes including source codes, thus, many malware authors tend to deploy those tools to deter malware analysis. Therefore, dealing with huge volumes of malwares requires an automatable approach to uncover hidden codes in packed binaries. To realize some degree of automation, a generic method should be considered thereby achieving feasible analysis on those huge volumes.

In this paper, we present a work-in-progress effort to realize generic malware unpacking. Our proposed method is basically to quantify the activities of stub code that is generally resided in a packed binary. A stub code is in charge of decompressing/decrypting; when the packed binary is executed, some sections that are compressed/encrypted by the stub. Therefore, we expect that if the stub code performs unpacking in runtime, some hidden (packed) code be revealed so that we can count the amount of the revelation. To realize the quantification, our architecture includes shadow states for process' memory space being instrumented: i.e., each byte of application memory is mapped to a unit in shadow memory. A shadow state maintains 1) a written byte is executed and 2) an executed byte is written. The two cases respectively reflect the newly revealed code by unpacking and disappearing code by possible re-packing. By managing shadow states, it may be possible to understand the behavior of stub code thereby achieving to find appropriate time – i.e., spotting candidate original entry points (OEP) automatically – to make process dump to be further statically analyzed.

Our proof-of-concept implementation is based on PIN, a dynamic binary instrumentation (DBI) framework developed by Intel [8]. Executing a packed binary, our tool tracks memory access activities and instruction executions to appropriately update shadow states. Our test results show that the proposed method is basically effective to find candidate OEPs for typical cases. Simultaneously, we also learned some limitations in our implementation: i.e., the memory integrity problem of DBI approach.

Paper organization. The remainder of this paper is organized as follows. In Section 2, we summarize related work and present the basic working of packed binaries as background. In Section 3, we describe our proposed method and proof-of-concept DBI-based implementation. Experiment results on some well-known packers are also included. We discuss the effectiveness and limitation of our approach in Section 4. We conclude this paper in Section 5.

2 Background

In this section we briefly overview related work and describe the basic working of a typical packer as background.

2.1 Related work

There have been considerable efforts for unpacking malwares in both research and industry. Technically, many of them are implemented as debugger plug-ins for IDA Pro [9], OllyDbg [10], and so on. On the other hand, works toward transparent instrumentation against malwares have been deployed system emulators (e.g., Bochs [11], QEMU [12]) or virtual machine monitors (e.g., Xen [13], VMWare [14]).

Eureka [7] is a static analysis framework of SRI and it includes an automatic binary unpacking unit based on heuristic and statistical method with system call granularity. Their simple heuristic is to make process snapshots at the program exit system call (NtTerminateProcess) and they commented that it works for incremental unpackers which gradually reveal hidden codes but never re-encrypt the once-revealed codes. They also track the NtCreateProcess system call considering that many malwares spawn their own images to evade naïve unpacking trials. Eureka also involves a statistical method that recognizes specific x86 code pattern bigram (prevalent patterns such as push or call instructions) frequency expecting that to be increased as stub unpacks.

OllyBonE [16] is an OllyDbg plug-in to support semi-automatic spotting OEPs exploiting page protection mechanism implemented as a kernel driver. A debugger user can manually selects some section in where stub likely extracts hidden code and transfers control there. At the time of detecting the control transfer targeting such page, the debugger automatically stops at the first instruction (i.e., a candidate OEP) with the help of page fault handler (trapped by int1).

Saffron [5] is proposed with a combination of two approaches (Saffron-di and Saffron-pfh). Saffron-di is based on PIN DBI framework. The implementation is very similar with our tool. Saffron-di tracks memory writings of a packed binary and checks whether current instruction pointer is within the previously written area in every execution. Saffron-pfh involves page-fault handler mechanism inspired by, thus similar with, OllyBonE. It is implemented by overloading page fault interrupt (0x0E) of Windows OS and managing supervisor bit for pages. OEP is spotted if previously written memory area (pages marked to be monitored) is executed in both approaches.

The approach of PolyUnpack [17] is based on the assumption that the instance of revealed code from a packed malware can be identified with the malware's static model. During a test run of packed binary, PolyUnpack performs simultaneous comparison between the current instance's instruction sequences and those of in the static model (extracted from the instance's binary). If some instruction sequences cannot be accorded with the static model, that portion is considered to include possible packed code.

Renove [24] monitors memory write activity and program execution with a modified version of QEMU (TEMU). Similar with our work, the architecture of Renove involves shadow memory to reflect written memory as dirty. If the instruction pointer jumps to some dirty area, the area is considered to contain some reveled code unpacked in runtime. Renove works with basic block level granularity.

Pandora's Bochs [18], built on Bochs x86 system emulator, uses dictionary data structure to store written memory (of recent 256 bytes) of user-mode processes. Whenever a branch instruction is about to be executed, the branch target is compared with the written memory to locate an OEP. Pandora's Bochs also includes Python

scripts for reconstructing PE header and import address table (IAT) to further process on dumped images.

Overall above works involve a similar approach: spotting executed memory area previously written W^X approach. However, there were no specific trials to observe the behavior of stub code by measuring the amount of revealed code.

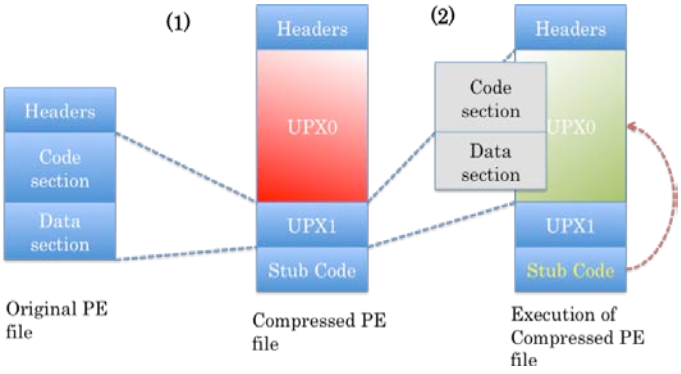

Fig. 1. The working of an UPX packed PE binary

2.2 Behavior of packed binaries

Here we describe the basic working of packed binary taking an example from a PE file with UPX packer [19]. UPX packer transforms a PE binary by compressing sections. After the compression phase [(1) of Fig. 1], the newly generated binary contains two sections named as UPX0 and UPX1. The original contents (code and data) are compressed into the UPX1 section with a stub code. The UPX0 section is virtually exited in the new PE header; thus, the actual file size becomes smaller than the original executable file. When the compressed binary is launched [(2) of Fig. 1], the stub code is firstly executed as the entry point of the original file is changed to point the stub code. The stub code decompresses the contents of UPX1 into the UPX0 section, actually allocated in memory, and then it transfers control to the uncompressed area thereby starting the original program context. The first branch target from stub to the area comes to the OEP.

The typical manual unpacking of a UPX packed binary file would involve finding OEP and is usually not hard to find it with a debugger or any other instrumentation tool. If one can find a branch to the UPX0 section where there is no meaningful code in static state, then the branch target would be the OEP. Making a breakpoint at the branch target, the unpacked code can be seen when the program is in break and the dump image of the unpacked state can be obtained with some additional post processing (adjusting PE header and reconstructing Import Table). Currently, many plug-ins in debugger support for those ability for manual unpacking to let analysts do their job in more comfort way.

3 Quantifying Code Revelation

In this section, we describe our design approach to realize generic malware unpacking. We present design and implementation of our approach and show some results of experiments.

3.1 Design

Our method is basically to leverage general behavior of stub codes dangled in packed binaries. Usually, a typical stub code in a packed binary takes the very first control of the execution and starts to unpack compressed and/or encrypted data in the memory (sections within the boundary of PE layout or possibly somewhere dynamically allocated area). The control is then handed over to the unpacked memory area. Therefore, the packed data eventually came out and our conjecture is that the unpacked data amount to revealed code.

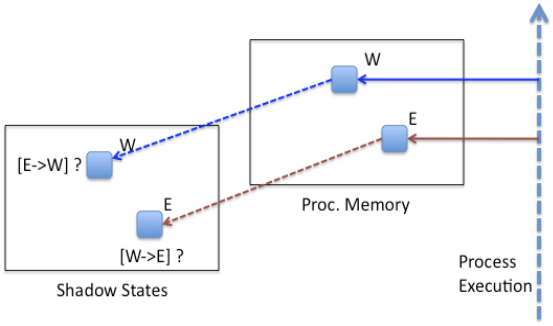

Fig. 2. The proposed design approach for quantifying code.

Our heuristic is to perform quantification of the code revelation shadowing user process memory. Fig. 2 depicts our design approach to realize for quantifying code revelation by reflecting memory accesses of a user-level process on shadow states. For a single byte in memory, if some data is previously written and then executed at the same byte, it amounts to the increase of a new byte code. To achieve this quantification, each state of shadow memory records the memory access activities, (specifically, write/execute for corresponding bytes) maintaining 4 possible states: i.e., [W] (write), [E] (execute), [W->E] (written byte is executed), and [E->W] (executed byte is written). State transitions occur according to the name of states expressed. Let r be the measure of revealed code. Managing memory access states during packed binary execution, the count of [W->E] increases r as the corresponding bytes of user memory contains newly uncovered executable instructions.

In accordance with the general W^X approach of manual unpacking, the area around the first [W->E] state comes out would be highly likely an OEP. After stub extracts and transfers control, normally the instructions of revealed code is

consecutively executed for a while and that would be measured with *r*. Therefore, our method to take (candidate) OEPs is of taking the first instruction pointer of a trace that turns suddenly increasing *r*.

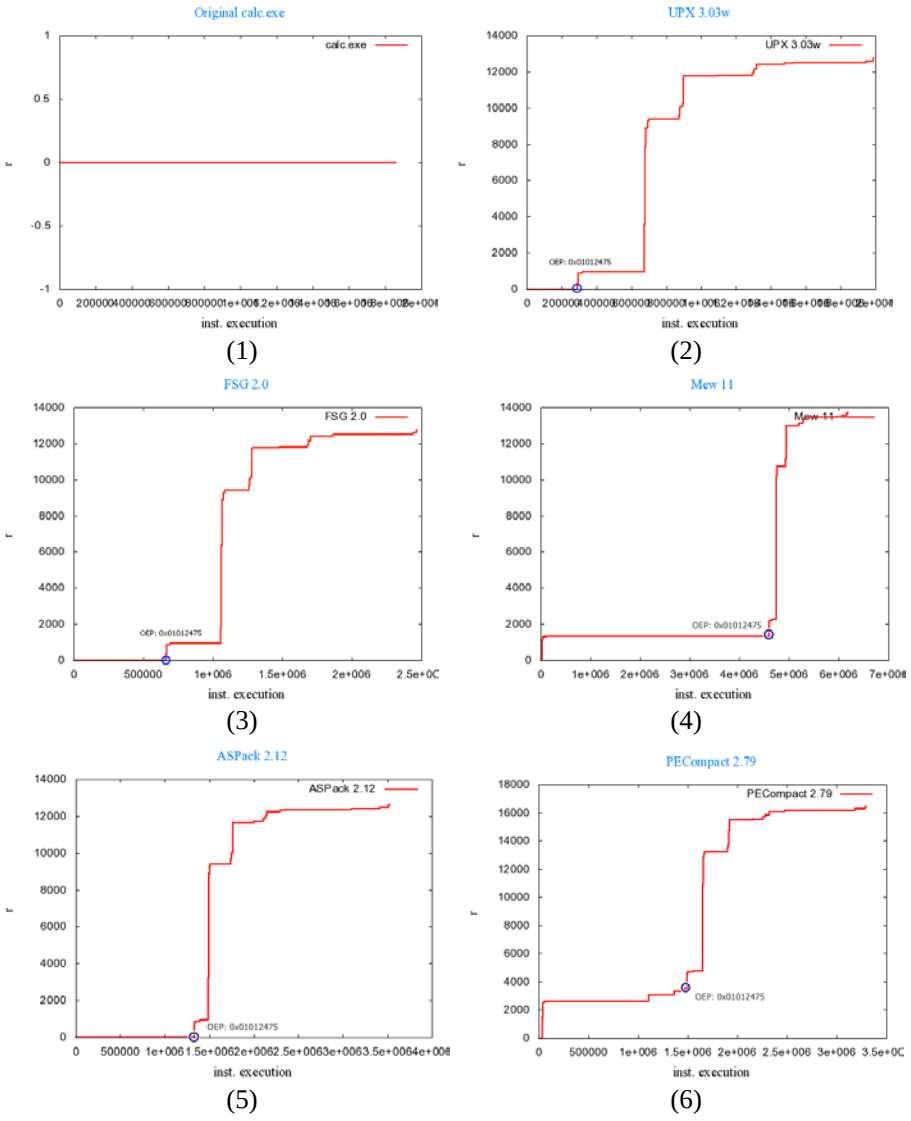

Fig. 3. Experiment results with calc.exe.

Meanwhile, if a state transition to [E->W] occurs, it will decrease *r*. The possible scenario of occurring this transition might be shown in the behavior of a packer armoring with *shifting decode frame* [5]. In this scenario, stub code might perform re-

packing a code portion after executing that portion which is previously unpacked. Although in the current work-in-progress status of our research, we only identified code revelation cases (increasing *r*) due to some limitation of our implementation framework which will be discussed in Section 4. Nevertheless, after solving the issue, we believe that our approach would be effective for identifying the behavior of packers that perform re-packing during the program execution to avoid being analyzed by naïve unpacking methods.

3.2 Implementation

Our method can be implemented on several types of instrumentation framework with which are possible to monitor memory accesses and track instruction pointers. Candidates are DBI frameworks (e.g., Valgrind [20], PIN [8], and DynamoRIO [21]) or system emulators/virtual machine monitors (e.g., Bochs [11], QEMU [12], and Xen [13]). Here we have implemented with PIN as it supports a rich set of manipulation APIs and we could quickly retrofit our previous implementation [22] of shadow memory for PIN usage. Especially PIN works in JIT(just-in-time)-based process instrumentation. The process execution under JIT-based instrumentation is known to be transparent compared with instrumentation of probe-based schemes [6].

Shadow memory should reflect the memory usage of user process's memory footprint. In many cases, only part of spaces among possible 2GB user process area are used; thus, we implemented shadow memory, for holding 4 states for each memory byte, with a simple 2 level page-table-like structure.

We added analysis functions to instrument memory accesses (write and execute) of every instructions and state transitions described in the previous subsection are occurred there. Additional facilities such as measuring *r* by counting current [W->E] states, and dumping current memory snapshot (the section including PE layout) are also included.

3.3 Experiments

We conducted tests to confirm effectiveness of our method with the basic calculator application (calc.exe, 112KB) included in Windows XP SP2. To determine sudden executions of revealed code, we assumed that 10-20 instruction window. The Figure 3 (1)-(6) illustrates our results for original calc binary and binaries packed with widely deployed in malware samples: i.e., UPX, FSG, Mew, ASPack, and PECompact, respectively. As it generates a quite large log traces, currently our tool logs the measure *r* with basic block granularity to plot graphs.

In Fig. 3(1), the behavior of original calc application does not show any revealing of new code. On the other hand, we can investigate some behavior of revealing new codes in the remaining graphs.

During the monitoring of packed binary executions, our tool generated a set of candidate OEPs for each case. Moreover, we could confirm that the each candidate set includes the exact OEP (0x01012475) in all cases. For the cases of UPX, FSG, and ASPack, the first addresses of their respective set were the exact OEPs [Fig.3 (2),(3),

and (5)]. For the cases of Mew and PECompact, the exact OEPs were at the 7th and 27th of amongst spotted OEP candidates [Fig.3 (4), Fig.3 (6)]. In both cases, however, we could discriminate the area without much difficulty as *r* increases rapidly around the spots. We also could obtain PE dump files for the OEP and those were successfully validated with a debugger and a PE analysis tool.

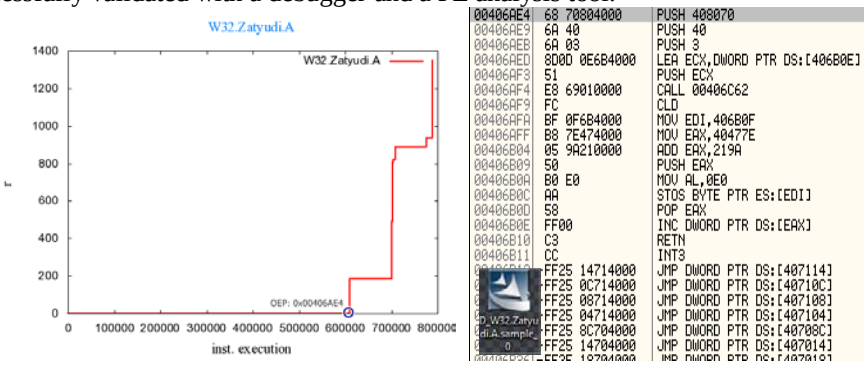

Fig. 4. Experiment with W32.Zatyudi.A sample

Our method is further tested with some real malware samples such as W32.Virut.B, W32.Zatyudi.A packed with PECompact, and ASPack respectively identified by PEiD. Figure 4 shows the test result with W32.Zatyudi.A worm virus sample and its dump of entry point. We could successfully compare with the OEP and the unpacked binary with the debugger result.

4 Discussion and Future Work

Here we discuss results and limitations of our current work-in-progress research and present future work to improve our work.

Our objective was to explore a method to support dealing with large volumes of malware samples that are possibly compressed/encrypted by packers to deter security analysis. The proposed method is basically relying on the behavior of stub code attached to packed binaries. Observing the sudden consecutive executions of instructions that are newly uncovered, i.e. measured as revealed code *r*, we conjecture that a candidate OEP could be found around the first instruction of a trace that sharply turns up the measure. Through the experiments, we could confirm that our approach is effective, henceforth; it is possible to be deployed for unpacking malwares as a generic method. The implemented tool spotted the exact OEPs automatically during the test executions of some packed binaries and the obtained unpacked PE binaries were manually identified to be correct. Among the automatically collected OEPs for a binary, we found some redundancies. We might reduce the size of candidate sets by eliminating redundant counts in shadow states.

As an instrumentation framework, DBI with JIT is known to be transparent in terms of instruction execution compared to Probe-based DBI [6]. This is because JIT-based DBI does not modify instruction code while the runtime code under probe-based DBI includes trampolines, resulting in the modification of original code

instructions, to be instrumented. However, we have a lesson learned about the downside of our implementation approach: with DBI, it is not feasible to be applied to some binaries that are packed with some tricky packers due to the memory integrity problem. A bit later, we could find comments of similar experience in the case of Saffron-di [5]. Even though the execution could be transparent, basically the memory layout between the executions of original context and DBI-driven are different because some additional memory sections, such as DBI engine or code cache, should be co-located in the same process boundary within the instrumented process. Accordingly, with some elaborated packers that perform memory integrity checking (e.g., CRC), DBI based implementation would not easily work.

Due to the limitation, we are now working on the same unpacking approach, moving to outside from user process layer, with a whole system emulator or a virtual machine monitor. For a long term, we plan to build a framework for malware unpacking in our working network incident analysis systems [4]. The framework would be a multi-phased including static identification, to quickly determine whether a given malware is packed or not, and dynamic analysis module to perform rigorous runtime analysis.

5 Conclusion

In this paper, we have presented our preliminary research efforts toward generic method to realize automatic binary unpacking concerning the large volume of malwares currently we encounter these days. Our method is basically to observe the specific tendency of stub code behavior in packed binaries by measuring the newly revealed code with shadow memory states architecture. We have implemented a proof-of-concept implementation based on PIN DBI framework and conducted experiments with the tool for some widely used packers. It is confirmed that our method is effective for dealing with binaries applied with those packers. On the other hand, we also found some downsides in the implementation and now are working toward to improve our research efforts using a whole system emulator.

References

1. T. Brosch, M. Morgenstern, “Runtime packers, the hidden problem?,” Presented in Black Hat Conference, 2006.
2. PEiD, http://www.peid.info/
3. R. Lyda, J. Hamrock, “Using Entropy Analysis to Find Encrypted and Packed Malware,” IEEE Security and Privacy, Vol. 5, No. 2, pp. 40—45, 2007.
4. D. Inoue, K. Yoshioka, M. Eto, Y. Hoshizawa, K. Nakao, “Malware Behavior Analysis in Isolated Miniature Network for Revealing Malware's Network Activity,” Proceedings of IEEE International Conference on Communications (ICC), pp. 1715—1721, 2008.
5. Quist, D., Valsmith, “Covert Debugging: Circumventing Software Armoring Techniques,” Black Hat USA 2007, 2007.

6. Vasudevan, A., Yerraballi, R. “SPiKE: Engineering Malware Analysis Tools using Unobtrusive Binary-Instrumentation,” Proceedings of 21th Australasian Computer Science Conference (ACSC 2006). 2006.
7. Sharif, M., Yegneswaran, V., Saidi, H., Porras, P., “Eureka: A Framework for Enabling Static Analysis on Malware,” Technical Report SRI-CSL-08-01. SRI International. 2008.
8. C. 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,” Proceedings of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), pp. 190–200, 2005.
9. IDA Pro, http://www.hex-rays.com/idapro/
10. OllyDbg, http://www.ollydbg.de/
11. Bochs, http://bochs.sourceforge.net/
12. F. Bellard, “QEMU, a fast and portable dynamic translator,” Proc. Of the USENIX 2005 Annual Technical Conference, FREENIX Track, pp. 41–46, 2005.
13. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP), pp. 164–177, 2003.
14. VMWare, http://www.vmware.com/
15. F. Guo, P. Ferrie, T. Chiueh, “A Study of the Packer Problem and Its Solutions,” Proceedings of the Recent Advances in Intrusion Detection (RAID). LNCS 5230, pp. 98—115, 2008.
16. OllyBonE, http://www.joestewart.org/ollybone/
17. Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, Wenke Lee, "PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware," Proc. of the 22nd Annual Computer Security Applications Conference (ACSAC'06), pp. 289—300, 2006.
18. L. Böhne. Pandora’s Bochs: Automatic unpacking of malware. Diploma thesis, RWTH Aachen University, Jan. 2008.
19. UPX, http://upx.sourceforge.net/
20. Nicholas Nethercote and Julian Seward, “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation.” Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), San Diego, California, USA, June 2007.
21. DynamoRIO, http://code.google.com/p/dynamorio/
22. Hyung Chan Kim, Angelos D. Keromytis, Michael Covington, and Ravi Sahita. "Capturing Information Flow with Concatenated Dynamic Taint Analysis," In Proceedings of the 4th International Conference on Availability, Reliability and Security (ARES), pp. 355 - 362. March 2009.
23. Jim Clausing's Malware Packer Signatures, http://handlers.sans.org/jclausing/userdb.txt
24. M. G. Kang, P. poosankam, and H. Yin, “Renove: a hidden code extrctor for packed executables,” Proc. of the ACM workshop on Recurring malcode, pp. 46—53, 2007.