An Implementation of a Generic Unpacking Method on Bochs Emulator

Hyung Chan Kim Daisuke Inoue Masashi Eto Jungsuk Song
Koji Nakao

National Institute of Information and Communications Technology
4-2-1 Nukui-Kitamachi, Koganei, Tokyo 184-8795 Japan

{hckim, dai, eto, song, ko-nakao}@nict.go.jp

Abstract In these days, it is very prevalent to discover many packed malwares caught in any malware collecting systems including honeypots. Thus, the initial step for usual malware analysis involves unpacking binary samples. In this paper, we present a yet another method of generic binary unpacking. A typical packed binary includes stub code that takes charge of unrolling packed data at the early stage of program execution thereby realizing original execution context. Our approach is basically to measure code revelation/concealment based on *byte state model* that reflects the behavior of such stub code. We also describe a proof-of-concept implementation based on *Bochs* x86 system emulator.

1 Introduction

One of the most pressing security concerns in recent years is to cope with malicious softwares (*malwares*). Those malwares, usually collected by honeypot systems, should be analyzed to establish any security countermeasures. However, large portions of such malwares we encounter these days are resistant to reverse engineering efforts: i.e., they are packed with any transformation methods such as compression, encrytion, and/or obfuscation to deter security analysis. Applying packing in malwares also hinders the detectability of anti-virus (AV) softwares [5] because a single known – thus its signature is alreay in our hands – malware sample can gain polymorhpism by being transformed by several types of packers.

An automatable generic unpacking method is required in two respects: (1) to perform the primary analysis of a large volume of malware samples; (2) to alleviate the burden of manual analysis.

Our research group has a collection of sample malwares size of 643,409 (as of May, 2009). Among them, at around 50% are identified as packed binaries with PEiD tool [3] based on a community supported signatures database [6]. As signature based identification often shows false negative results, we also conducted an entropy based identification [11], naïvely setting thresholds (6.7 for entire file and 7.2 for highest section), and it resulted in that the 90% of samples outnumbered the thresholds. This tendency comes from the publicly available packing tools, sometimes even including source codes; thus, a large number of malwares are deemed to be packed to avoid being analyzed. Therefore, manually dealing with numerous samples, security analysts cannot bear the whole load of analysis.

In this paper, we propose a yet another generic unpacking approach. Our approach is basically to measure code revelation/concealment, shadowing the behaviors of typical stub code, during execution of packed binary. To achieve the measurement, we extend a general unpacking heuristic with a simple finite state model (*byte state model*). Our architecture includes shadow memory to maintain states of memory access activities.

We also describe a proof-of-concept implementation based on *Bochs* x86 system emulator [1]. Executing a packed binary sample inside the emulator, our module tracks memory ac-

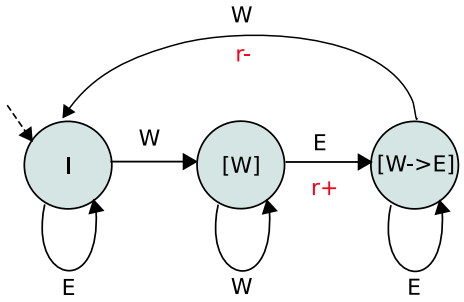


Figure 1: Byte state model

cess activities including instruction pointers to appropriately update shadow states by which code revelation is measured and candidate *original entry points (OEP)* are spotted. Our test results show that the proposed method has efficacy for many typical packers.

2 Design and Implementation

This section presents a design and proof-of-concept implementation of a generic unpacking approach.

2.1 Design

In many cases, packed binaries include *stub code* that unrolls packed portion on the fly. When a typical packed binary is executed, the attached stub takes the very first control[1], thus it starts to unfold (decrypt or decompress) the packed portion into memory. After completing the unfolding, the stub then hands over program control to the newly unfolded area thereby realizing original program execution. The first branch target, located by the stub, in the area could be a possible OEP.

The proposed method is basically to leverage the above general behaviors of stub code. To reflect the behaviors, we introduce a simple finite state model [Fig. 1] for every bytes of native memory that is used by target process. Our heuristic is to perform quantification of the code revelation according with the state model: let r be the measure of code revelation, if there is a newly discovered byte, r is increased. In accordance with the general heuristic [9, 4, 8], if a byte is previously written and then executed at the same byte, it amounts to revealing a new code. To achieve this quantification each state represents memory access activities: other than the initial state $[I]$, $[W]$ (write), and $[W \rightarrow E]$ (written byte is executed). State transitions can be occured with memory write instruction (W) and every instruction execution (E). Managing memory access states during a packed binary execution, the count of $[W \rightarrow E]$ states over process memory range represents the newly unrolled code area. Meanwhile, if a process writes something to the address range associated with $[W \rightarrow E]$ states, it means code concealment: the once executed code are eliminated from memory. The possible scenario of occuring this transition might be *shifting decode frame* [12]. With this technique, a stub may perform possible repacking that results in overwriting some bytes on the already executed area.

The original execution context is started after unrolling, in where many writing operations are involved, packed portion. Thus, we might be able to observe abrupt change of the measure r at around the OEP. Our conjecture is that r would begin to sharply increase at the plausible OEPs as many state transitions from $[W]$ to $[W \rightarrow E]$ would occur after the turning points. We try to find such turning points to spot candidate OEPs during execution. However, because packed binaries basically involve code-modifying characteristic, there may be some noises (false OEPs) posing similar phenomenon. More sophisicated binaries would result in many such points. Therefore, we collect all the such points and try to generate a candidate OEP set of feasible size.

2.2 Implementation

Figure 2 depicts our implementation architecture. We use *Bochs* x86 whole system emulator (Ver. 2.4.1) [1] to instrument the behaviors of packed binaries. We choose Bochs as it provides an easy way to instrument fine-grained memory accesses, control flows, as well as instruction executions; those are requirements of our design approach.

[1] Usually, in a packed executable, entry point in the program header points to stub code area.

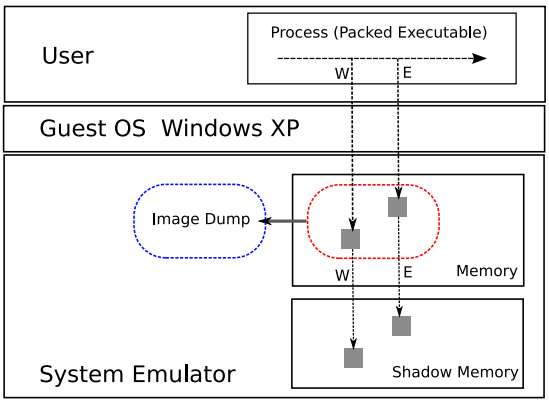


Figure 2: Architecture

On top of the system emulator, a guest OS (Windows XP SP2) is installed. Compared with process-based instrumentation approaches, such as deploying dynamic binary instrumentation (DBI) or using debugging APIs, it is necessary to parse internal data structures of Windows OS [13] to only keep track of concerned processes (e.g., memory area and instruction executions associated with a specific *CR3* register value). If a target process is started, initialization status and instruction pointer of the entry point of the process are checked. Validating those, fine-grained memory tracking, to quantify code revelation and concealment, begins.

Our implementation includes shadow memory to populate byte states of every native bytes of a monitored process: 1 byte in process memory is associated with 2-bit unit thereby holding up to 4 states. The shadow memory is implemented based on a page-table-like structure, allowing us to scale memory requirments with the actual process address space in use. Because we adopt 1 byte precision, the measurment of code revelation depends on the size of instructions.

During instrumentation, our module generates a candidate OEP set according to the heuristic described in Sect. 2.1. Currently, we use a window-based method: setting an instruction window, it is checked that whether past instructions within the window increase code measure r over some threshold. If exceeded are *branch* instrunctions within the boundary of binary image area, those instruction pointers are emitted to the OEP set.

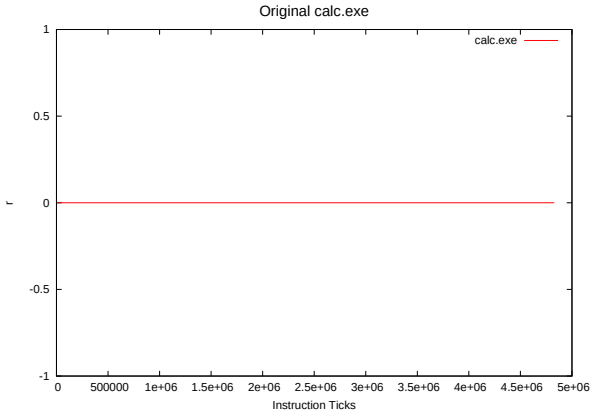
To make image dump files of the monitored process (e.g., sections including PE layout) at the appropriate candidate OEPs, we exploit frame page faults for the target memory area. Sometimes, especially at the early stage of program execution, some parts are missing within the physical memory as the underlying OS does not load some page frames until those are really necessary; thus, we need to load corresponding page frames into memory.

3 Experiments

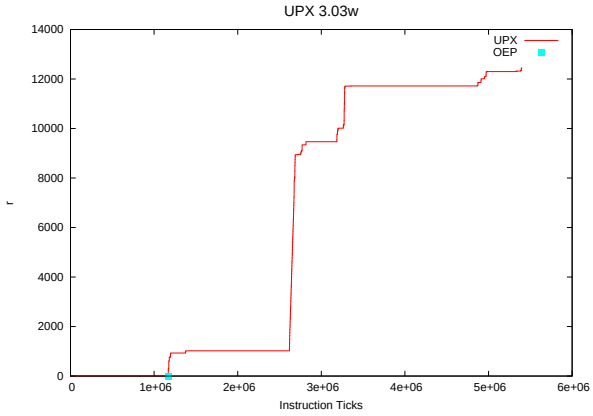
We conducted tests to validate efficacy of our unpacking approach with the calculator application (`calc.exe`, 112KB) included in Windows XP. The sample binaries were generated by packing the calculator with various packers available in the Internet.

3.1 Test Results

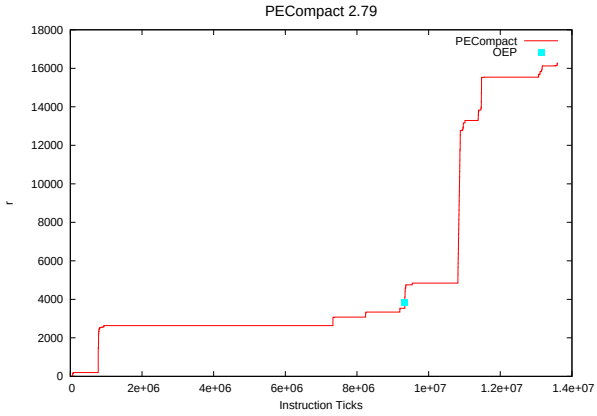
Figure 3 shows the quantification results of original binary and binaries packed with some widely known packers: UPX, PECompact, Yoda Crypter, ASPack, and PESpin. The graphs show the variation of revealed/concealed code measurement with instruction precision (except the PESpin, it is plotted with branch precision), and the OEP of calculater program (0x1012475) is superimposed on the surfaces. As shown in the Figure 3(a), there was no variation in the execution of the original binary as it dose not accompany any code-modifying behavior. Meanwhile, the other packed binaries exposed their own specific unrolling behaviors. Observing these graph results, and the results of some other packed binaries which are not included here, executions of test samples exhibited staircase behaviors in regards to memory accesses. Flat phases may involve heavy read and write accesses and bump increase phases indicate heavy executions of newly exposed instructions (i.e., state transitions from $[W]$ to $[W \rightarrow E]$). Most sample runs resulted in non-decreasing behaviors as new code rolled out by the attached stub code of each binary. From the observation, we conjecture that the initial



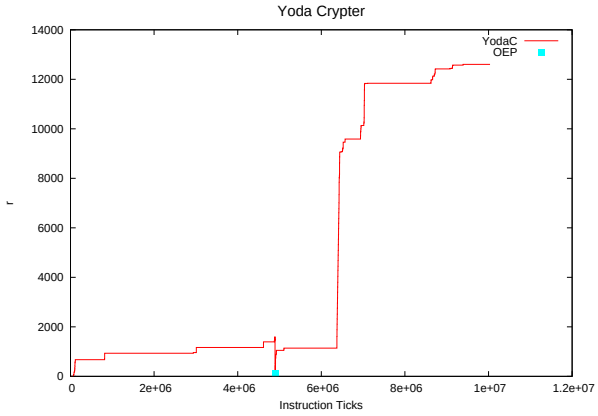
(a) Original Calc.exe



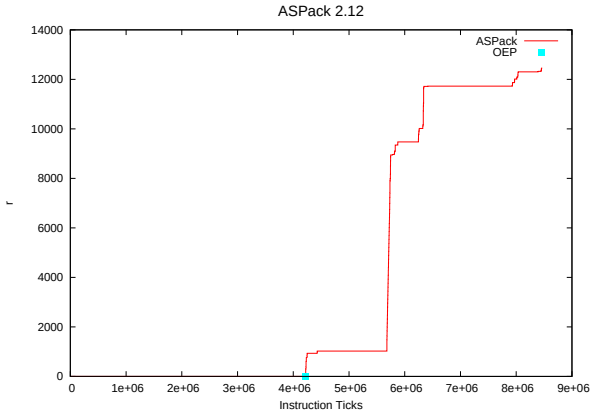
(b) UPX



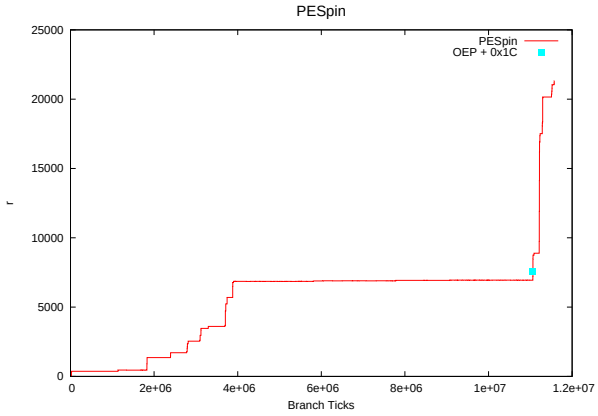
(c) PECompact



(d) Yoda Crypter



(e) ASPack



(f) PESpin

Figure 3: Experimental results: measuring code based on byte state model

behavior of packed binaries are well accorded with the general heurstic. Moreover, we also observed that the OEPs were located in turning points: i.e., the points in where a sudden phase shift occur.

However, there were some exceptional cases. Yoda Crypter exhibited code concealment behavior just before landing at the OEP [Fig. 3(d)]. To verify the concealment, we generated an instruction execution trace annotating write accesses, which contribute to decrease code measure, and then compared with a debugging result. The concealed area, which is based on our model and actually the area was filled with zeros, was proved to be coincided with the newly written area at the very early of the execution (even before the unrolling main executable code). This explains that the stub is generated dynamically and removed later.

With the PESpin case [Fig. 3(f)], the exact address of the OEP was failed to be located. Instead, the address 28 bytes apart from the OEP was spotted. This is because an obfuscation technique, *stolen bytes* [14], is applied to obstruct analysis: initial bytes around the OEP were moved to another area, where also appears to be obfuscated, and the original place was filled with zeros.

Table 1 shows the unpacking test results of packers easily obtainable from the Internet. In the table, the first column is the size of generated OEP set; whether the generated OEP set includes real OEP is marked in the second column; the third column is the obtainability of apporpriate image dump files.

In most cases, our system generated feasible size of candidate OEP sets with an optimization: if a candidate OEP is spotted, we reset continuous area of same state to initial state in shadow memory. The resulted sets mostly include the exact OEP. Cases involved with obfuscation such as stolen bytes, the exact OEP was not included in the generated sets. In our test, the nearst locations spotted were apart by 7 bytes in Obsidium and PELock cases, and by 28 bytes in PESpin.

Our system failed to spot OEP and to obtain valid dump file from executables packed with Telock and VMProtect. Telock performed memory integrity (CRC) check and our system was detected. The further analysis was halted by the detection with a warning message. The binary packed with VMProtect, which deploys virtual machine techniques, even did not generated any candidate OEP.

Table 1: Unpacking results

Packer	Set Size	OEP	Dump
ASPack	3	O	O
ASProtect	21	O	O
eXPressor	3	O	O
FSG	3	O	O
MEW	5	O	O
MoleBox	5	O	O
Morphine	5	O	O
npack	3	O	O
Obsidium	65	X	O
PACKMAN	4	O	O
PECompact	5	O	O
PELock	16	X	O
PEPack	3	O	O
PESpin	11	X	O
petite	6	O	O
RLPack	3	O	O
telock	3	X	X
UPX	3	O	O
VMProtect	0	X	X
WINUPack	7	O	O
yC	4	O	O
yP	5	O	O

3.2 Discussion and Future Work

As shown in the test results, our system, implemented with a general approach, works for many typical packers. Moreover, it is possible to understand the behavior of packed binaries with the generated graphs based on byte state model. As the resulted OEP set sizes are not too large, dumped images at those OEPs might be feasibly considered to be input for further static analysis.

However, we also identified limitations. First is related with the transparency issue of underlying instrumentation framework. In our

experiment, we could not execute the binary packed with Telock for analysis. This might be because Bochs does not perfectly emulate execution contexts compared with native environments. For further investigation, actually, we also have implemented our method in a *Xen* based instrumentation framework, *Ether* [7], which also supports fine-granined memory access traces and is known to be more transparent as guest OSs are executed near natively under *Intel VT* [2] support. For Telock case, we could successfully generate candidate OEPs and obtain valid dump files on that framework.

Second, binaries protected with virtual machine techniques do not exhibit code-modifying feature (as VMProtect case in our test); thus, the general heuristic of detecting executed bytes of previously written may not work. Instead of unrolling native code by stub, virtual machines are included in those binaries. Protected code (*p-code*), different with native x86 instructions, are interpreted and executed by the attached virtual machines.

Our work is currently in its preliminary status. For functional enhancement, we will include modules (1) to make dump not only image area but also dynamically allocated area, and (2) to reconstruct import address table. We also will continously deal with several instrumentation frameworks or try with methods to cover emulation bugs [10].

4 Conclusion

In this paper, we have presented our work-in-progress efforts to build toward a generic and automatable unpacking system thus performing the primary analysis of a large volume of malwares. Our method is basically to quantify newly revealed/concealed code with byte state model. We also presented a proof-of-concept implementation based on Bochs whole system emulator as well as test results thereby confirming the efficacy of our approach.

References

[1] Bochs ia-32 emulator project. `http://bochs.sourceforge.net/`.

[2] Intel virtualization technology. `http://www.intel.com/technology/virtualization/`.

[3] Peid. `http://www.peid.info/`.

[4] L. Böhne. Pandora's bochs: Automatic unpacking of malware. Master's thesis, RWTH Aachen University, 2008.

[5] T. Brosch and M. Morgenstern. Runtime packers, the hidden problem. Black Hat Conference, 2006.

[6] J. Clausing. A packer signature for peid. `http://handlers.sans.org/jclausing/userdb.txt`.

[7] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62, New York, NY, USA, 2008. ACM.

[8] F. Guo, P. Ferrie, and T.-C. Chiueh. A study of the packer problem and its solutions. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 98–115, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] M. G. Kang, P. Poosankam, and H. Yin. Renovo: a hidden code extractor for packed executables. In *WORM '07: Proceedings of the 2007 ACM workshop on Recurring malcode*, pages 46–53, New York, NY, USA, 2007. ACM.

[10] M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song. Emulating emulation-resistant malware. Technical Report UCB/EECS-2009-58, EECS Department, University of California, Berkeley, May 2009.

[11] R. Lyda and J. Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security and Privacy*, 5(2):40–45, 2007.

[12] D. Quist and Valsmith. Covert debugging: Circumventing software armoring. Black Hat Conference, 2007.

[13] M. E. Russinovich and D. A. Solomon. Windows internals (5th edition), 2009.

[14] M. V. Yason. The art of unpacking. Black Hat Conference, 2007.