IEICE TRANS. INF. & SYST., VOL.E94–D, NO.9 SEPTEMBER 2011

PAPER

An Empirical Evaluation of an Unpacking Method Implemented with Dynamic Binary Instrumentation**

Hyung Chan KIM[†*a)], ***Member***, **Tatsunori ORII**[††], ***Nonmember***, **Katsunari YOSHIOKA**[††], **Daisuke INOUE**[†], **Jungsuk SONG**[†], **Masashi ETO**[†], **Junji SHIKATA**[††], **Tsutomu MATSUMOTO**[††], ***and*** **Koji NAKAO**[†], ***Members***

SUMMARY Many malicious programs we encounter these days are armed with their own custom encoding methods (*i.e.*, they are packed) to deter static binary analysis. Thus, the initial step to deal with unknown (possibly malicious) binary samples obtained from malware collecting systems ordinarily involves the unpacking step. In this paper, we focus on empirical experimental evaluations on a generic unpacking method built on a *dynamic binary instrumentation (DBI)* framework to figure out the applicability of the DBI-based approach. First, we present yet another method of generic binary unpacking extending a conventional unpacking heuristic. Our architecture includes managing shadow states to measure code exposure according to a simple byte state model. Among available platforms, we built an unpacking implementation on PIN DBI framework. Second, we describe evaluation experiments, conducted on wild malware collections, to discuss workability as well as limitations of our tool. Without the prior knowledge of 6029 samples in the collections, we have identified at around 64% of those were analyzable with our DBI-based generic unpacking tool which is configured to operate in fully automatic batch processing. Purging corrupted and unworkable samples in native systems, it was 72%.

key words: *software security, dynamic binary instrumentation, unpacking, malware, binary analysis*

1. Introduction

One of the most pressing security concerns in recent years is to cope with malicious software (*malware*). Unfortunately, a large portion of malware specimens are resistant to binary analysis: namely, many of the specimens are *packed* [1] with transformation methods such as compression, encryption, and/or obfuscation. Apparently, it becomes getting more difficult to analyze packed malware. Even worse, already known malware can be morphed into other forms by packing thereby hindering the detectability of anti-virus (AV) software [2].

An analyst may perform *manual unpacking* for a single sample or couple of highly intricate samples controlling unpacking sessions minutely. Meanwhile, if there are numerous samples to be processed primarily, some heuristic methods, devised to be applied to packed binaries without any prior knowledge of specific packers, might be implemented as generic unpacking tools to realize automatic batch processing. Either method can be *static* or *dynamic*: *i.e.*, unpacking procedure can be processed based on static information of sample files or on dynamic contexts obtainable by directly executing them.

Several types of *virtualization* platforms have emerged as deployable dynamic binary analysis framework over recent years. Such platforms are extensively used for isolation in building security solutions. At the same time, some platforms enable us to perform *instrumentation* and/or *introspection* of given guest units for dynamic analysis. Different virtualization techniques suggest different viewpoints for security engineers. Among them, virtual machine monitors (VMM) (*e.g.*, VMware [3], Xen [4], and KVM [5]) and whole system emulators (*e.g.*, Bochs [6] and QEMU [7]) provide system-wide view inspection. Meanwhile, some *dynamic binary instrumentation (DBI)* frameworks (*e.g.*, PIN [8], Valgrind [9], DynamoRIO [10], StarDBT [11], and DynInst [12]) support ways to instrument with process-wide view.

Using these platforms, ensuing research efforts have been made to build dynamic analysis tool, including unpacking, reckoning on the realizable instrumentation abilities of each platform. With the system-wide virtualization platforms, unpacking modules or tools have been implemented with VMMs [13], [14] and system emulators [15]–[17]. On the other hand, unpacking tools based on DBI frameworks [18]–[20] also have been developed in parallel. Although, VM platforms have been actively considered likewise, it is not easy to find some referential data regarding usability. In other words, many research works have suggested comments or discussions about platform usability and/or limitation, there were few attentions on sole framework evaluation to look into overall workability testing with a substantial size of malware samples. Among deployable VM frameworks, this paper explores the applicability of unpacking implementation based on process-wide DBI framework as an empirical case study to assist deployment decisions of analysis platforms.

To implement a target unpacking tool, to be used in our case studies, first we present a design of an unpacking method. Our approach, an extension of common unpacking heuristic, is to measure code revelation as well as concealment behavior during executions of packed binaries: *i.e.*, if new code bytes are discovered (or known to be removed),

Manuscript received August 11, 2010.
Manuscript revised January 30, 2011.

[†]The authors was and are with the National Institute of Information and Communications Technology (NICT), Koganei-shi, 184–8795 Japan.
[††]The authors are with the Yokohama National University, Yokohama-shi, 240–8501 Japan.
[*]Presently, with the Attached Institute of ETRI, Daejeon, 305–600 Korea.

[**]Primary version of this paper were presented at the 4th and 5th Joint Workshop on Information Security (JWIS).

a) E-mail: hyungchan.kim@gmail.com

DOI: 10.1587/transinf.E94.D.1778

code measure is increased (or decreased) accordingly. These activities are detected by maintaining shadow memory and each unit of shadow states is accorded with byte state model which is devised to reflect code revelation/concealment. The measure is determined by quantifying states which represent revealed (thus, unpacked) bytes. The graph resulting from the code measurement is helpful to understand unpacking behavior. The tool is implemented on PIN DBI framework [8] which is widely used among research works, not just for unpacking but also for other analysis purposes. Therefore, we believe that our work would be helpful for whom consider to deploy PIN DBI for their works.

For the applicability exploration, this paper includes empirical evaluation results of our tool. Our first evaluation is to perform unpacking jobs for tool efficacy test against several packers. Packed variations generated from 20 packers were prepared and then unpacking sessions were performed. Code section similarities were calculated among the unpacked binaries to check unpacking results. The second evaluation experiments were performed on wild malware collections of two spanned periods. To investigate magnified as well as grouped aspects, the experiments were divided into one-day and spanned periods. Measuring unpacked code can be achieved with calculating specific bigrams which reflect program control flows. Without the specific prior knowledge of 6029 samples collected within two periods, we have identified that 64% (overall) or 72% (purging corrupted and unworkable samples in native analysis system) workability with the DBI-based generic unpacking approach by configuring our tool to be worked in an automatic batch system realized with VMware.

Paper Organization. Section 2 briefly presents DBI concept and explores related work on unpacking research efforts. In Sect. 3, we describe our design and implementation of an unpacking tool based on a DBI framework. To figure out efficacy and applicability, Sect. 4 reports experimental evaluation results. We discuss our work in Sect. 5. Section 6 concludes this paper.

2. Background

2.1 Dynamic Binary Instrumentation

Binary instrumentation is a program transformation method that can enable to inspect or modify program execution contexts of a running program. Usually, transformation for instrumentation is embedding some code snippets to call analysis functions in where one can perform code context manipulation (*e.g.*, inspection and/or modification) (Fig. 1). Basically, there are two instrumentation approaches: static binary instrumentation (SBI) and dynamic binary instrumentation (DBI) [21]. Before a target program execution, we might be able to statically patch the program binary so as to let the program calls our analysis functions at our intended moments (*e.g.*, every time before `ret` instruction invocations). Such approach is in line with SBI. The patched,

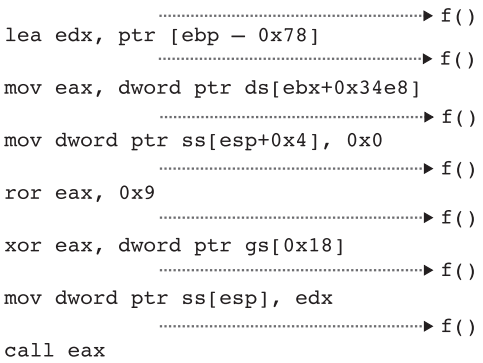


Fig. 1 Instrumented code block example: The inserted analysis functions provide manipulation (inspection and/or modification) moments of a running program.

thus instrumented, program might be executed in *native mode* with overhead incurred by the patched routine only. However, it is hard to instrument for unresolved execution flows (*e.g.*, code area reachable by indirect jump or self modifying behavior) as those might be not decided before execution. Meanwhile, with DBI approach, it is possible to dynamically arrange to direct to call analysis functions at runtime. Therefore, if appropriately handled at runtime, the statically unresolved code area can be covered for instrumentation†. However, setting aside analysis routine overhead, DBI often requires runtime arrangement which brings additional non-negligible performance overhead. As one of dynamic program analysis approaches, DBI is well accepted in fine-grained (instruction or basic block level) program analysis such as unpacking or dynamic taint analysis.

DBI can be realized with software virtualization techniques such as dynamic interpretation or just-in-time compilation (JIT). For example, Bochs [6] performs whole system emulation by interpreting one instruction at a time. Therefore, one can easily inject analysis functions for every instruction executions. Bochs actually supports a plug-in interface for dynamic instrumentation. Many other frameworks or emulators perform block based code translation to enhance performance compared to interpretation. Valgrind [9] and QEMU [7] perform basic block based translation: at a time, a basic block is translated into the corresponding intermediate block and then compile it with host instruction set. Meanwhile, PIN [8] and DynamoRIO [10] directly translate blocks without involving intermediate representations. In either methods, one can inject analysis functions by intervening block translations. Note that while PIN, Valgrind, and DynamoRIO work with single process boundary, Bochs and QEMU perform whole system emulation; thus, one should choose appropriate framework considering analysis scenario.

It is also known that fine-grained DBI is possible with hardware-assisted virtualization approaches such as with KVM [5] and Xen [4]. For example, Ether [13] over Xen

†Note that not all paths are covered. Only executed paths can be explored.

realized fine-grained instruction and memory write tracking exploiting debug exception and page fault mechanisms. However, whenever trap occurs, handler (analysis function) invocation involves VM context changes which incur severe performance overhead.

Overall, DBI with above frameworks/virtualizers have advantages in terms of execution context transparency, compared to debugging technique. Conventional debuggers involve debug trap instruction (*e.g.*, `int 3`) or set some debug specific contexts such as debug flags or designated debug registers. On the other hand, DBI does not aggressively involve such target-viewable context modifications. However, process-wide DBI frameworks do not fully guarantee memory transparency as some components may be co-resided with processes under instrumentation.

2.2 Related Work

In recent 5 years, there have been considerable efforts for developing unpacking strategies as well as tools on various platforms.

Sun et al. [22] developed a plug-in for `IDA Pro` disassembler to approximately spot original entry points (OEP). Their method is to record frequency of executed instructions and sort resulted histogram by the last time an instruction is executed. Then OEP may be found at around flat area after some spikes in sorted histogram as OEP is usually executed only once and unpacked area is usually not previously executed.

Eureka [23] static analysis framework includes a dynamic unpacking unit implemented as a Windows device driver which hooks system call referring SSDT (System Service Dispatch Table). Concerning incrementally packed binaries, their heuristic is to make dump files at `NtTerminateProcess` and `NtCreateProcess`. Eureka also involves a statistical method that recognizes the frequencies of prevalent x86 code patterns which may vary as an unpacking session progresses.

OllyBonE [24], OmniUnpack [25], Saffron-PHF [18], and Justin [26] have similar core architectures in that these works exploit page protection mechanisms in Windows OS and CPU support in some cases: *i.e.*, enforcing strategies similar with DEP (Data Execution Protection) with the help of NX (NoExecute) bit by hardware support or custom implemented PaX-like kernel layer driver modules. OllyBonE is incorporated with `OllyDbg` debugger and Justin is connected with anti-virus software.

The approach of PolyUnpack [16] is based on the assumption that the instance of revealed code from packed malware can be identified with the static model of it. During a malware sample execution, PolyUnpack performs simultaneous comparisons between the runtime instruction sequences and those of in the static model extracted from the sample file: namely, *self-identification*. On a failure of comparison, the differentiated portion is considered as possible packed code. Josse's work [27] also performs, like PolyUnpack, self-identification to expose hidden code: the modified QEMU emulator performs binary differentiation comparing runtime (translated) block resided in virtual memory and corresponding values in the file system.

Renovo [15], a plug-in of BitBlaze [28] binary analysis platform, monitors memory write and execution activities with shadow memory, which is similar with our architecture. With the memory shadowing, Renovo checks written memory as dirty. When program counter points to some dirty area, the area is considered to contain hidden code. Saffron-DI [18], Pandora's Bochs [17], and the unpacking tool included in Ether [13] also take very similar approaches with Renovo using hash table-like data structure to keep track of written memory area and detect attempted executions on there by monitoring branch targets.

Christodorescu et al. [29] proposed formal algorithms for transforming malware, including unpacking transformation, into obfuscation-free (unpacked) binaries. The transformation is based on the *program exposure oracle* which provides materialized control flow information by executing packed binaries. Detecting such control flows involves monitoring memory area where was previously written and then execution is attempted.

TraceSufer [19] provides abilities to profile behavioral properties of self-modifying programs, representatively packed binaries. Their method is to maintain dynamic types – individual type state is consist of (r, w, x) – for each memory addresses and give answers whether given packed binaries involve decryption, integrity check, and/or scrambling based on their criteria on memory access behavior.

SD-Dyninst [30] was proposed manifesting a phased analysis strategies for realizing hybrid (static and dynamic) analysis of analysis-resistant, including packed, binaries. The phases include initial binary parsing to obtain statically analyzable control flows. Dynamic analysis phases then follows to capture previously non-reachable control flows as well as code overwriting behavior.

Bania developed MmmBop [20] DBI framework specially considering anti-debugging and anti-reverse engineering tricks. MmmBop is equipped with several artifacts for unpacking dealing with cautionary issues such as `call` instruction instrumentation, and handling self-modifying code. For OEP finding, MmmBop monitors control transfers and assumes that if a control target lands to the border of first section of a packed file, it is generally considered as an OEP candidate.

We could find one work that draws our interest is the evaluation result conducted by Reynaud [31]. Instead of unpacking test, he used his PIN DBI-based tool to know how many malware, in his repository, applies anti-virtualization techniques.

Platform deployment. PolyUnpack, Renovo, the works of Josse and Christodorescu et al. are built on or involve QEMU system emulator [7]. Pandora's Bochs is built on Bochs emulator [6]. Ether-unpack depends on Xen [4]. Meanwhile, there are works based on DBI approaches: Saffron-DI and TraceSufer are developed with PIN DBI [8]

and SD-Dyninst works with DynInst [12]. MmmBop is based on its own custom DBI implementation.

Common unpacking heuristic. Broadly speaking, most of the above works follow the well-known packer principle, *i.e.*, a packed binary should be unpacked itself when it is executed. Therefore, the behavior necessarily involves control branching to newly revealed memory area in where unpacked code may be resided. In accordance with this principle, many works deploy *common unpacking heuristic* [26], and that is enforcing the detection of memory locations previously written[†].

Basically our unpacking method, for the case study tool implementation, extends the common unpacking heuristic. In terms of system architecture, our approach is similar with Renovo [15], Pandora's Bochs [17], Saffron-DI [18], and Ether-unpack [13] as these implementations also involve memory shadowing facilities to detect hidden code layers. However, our tool also measures code revelation/concealment typing memory states. The primary objective of this work is to present evaluation data testing with a rather large set of wild malware samples with a DBI-based analysis tool to figure out framework applicability. Saffron-DI also implemented with PIN DBI; however, they presented results based on a small set of samples. The other works are implemented on different platforms other than process-wide DBI framework.

3. An Unpacking Tool Based on DBI Framework

In this section, we describe our design approach to realize generic binary unpacking. In addition, an implementation of the design is presented.

3.1 Design Description

Our design approach for building unpacking tool is to be generic in conformity with the packer principle: our assumption is that no prior knowledge of specific packers, including possibly obtainable results by performing pre-stage static analysis, is required.

In many cases, a packed binary includes a small code (*stub code*). The first address of this code is designated as *program entry point* (PEP) in PE[††] header so that the stub code can be executed at the very first of program execution. The stub code then starts to unpack packed (*e.g.*, compressed, encrypted, and/or obfuscated) data, into another portion of memory. On the completion of the unpacking, the stub hands over program control to the newly unfolded code thereby starting original program execution. The first branch target in the new area usually coincides with OEP: *i.e.*, the possible original PEP of the not-yet-packed binary or the first address from where main program behavior is regularized after the end of unpacking.

The method is straightforward with the above mentioned behavior: our approach is to observe the pattern of new code exposure resulted from unpacking behavior of stub code. With this information, it is possible to understand the unpacking behavior of the target process executed from a packed file as well as to decide when make dump files. To reflect such behavior, we perform code measurement by quantifying how much code is newly exposed.

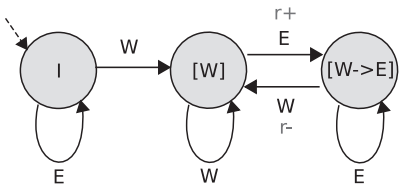


Fig. 2 Byte state model: We denote the state transition which write a value to a certain memory byte as `W`, and every instruction executions is considered as `E`.

To realize such measurement, we introduce a simple finite state model (Fig. 2) which is associated with every bytes of target process memory. Let r be the code measure. The measure will be increased if there is a newly discovered code byte. In accordance with the common unpacking heuristic [26], if a byte is previously written and then executed on the same byte, it amounts to exposing that (new) code byte. To achieve this quantification, each state of all the bytes represents concerning memory access activities: other than initial state $[I]$, $[W]$ (written), and $[W \rightarrow E]$ (written byte is executed). The write (`W`) transitions can be occurred with x86 instructions which write values to some memory bytes (*e.g.*, `mov [edi], eax`). Every instruction executions are considered as `E` transitions. Managing memory states with byte state model for all the bytes of target process, we assume that the count of $[W \rightarrow E]$ states represents the amount of exposed code. The counting is performed with taking in- and out-going transitions to and from the states.

Meanwhile, it is not only that the code measure r ever-increasing but also that the measure can be decreased if stub code write values to the address range associated with $[W \rightarrow E]$ states. Possible scenario is *repacking* or code elimination to deter static analysis against dumped files.

For at a given byte, it might be possible that a write instruction is executed as well as it writes a value to the same byte simultaneously. It would be rare case, but not impossible in *self-modifying code (SMC)*. In this situation, we adjust that the two transitions occur in the sequence of `E` followed by `W` assuming that the written is the result of the write instruction execution. After all, only enforcing `W` transition in such case is enough in byte state model.

To enforce byte state model for all the bytes of target process, we involve shadow memory architecture [33] to maintain state information during execution (Fig. 3): each byte in native memory has its own individual cell to be associated with a specific state at a certain time.

[†]It is also in line with enforcing the `W^X` policy as OpenBSD [32].

[††]An executable file in Windows OS is accorded with *Portable Executable (PE)* binary layout.

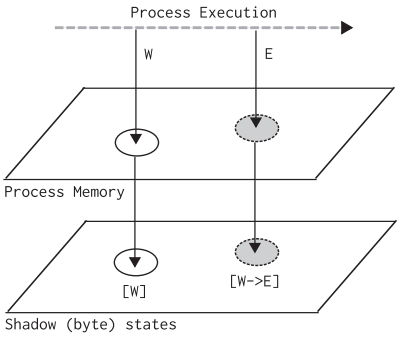


Fig. 3 Quantifying code exposure with shadow states.

Some malware specimen tries to elude being dumped in a way that unrolling code in dynamically allocated memory area. This usually deviates from the speculation of debuggers. To deal with this problem, our tool also enforces another heuristic to obtain exposed code: code executions occurred outside of main PE executable and associated DLLs are logged and dumped.

3.2 Implementation

We have implemented a target unpacking tool based on PIN DBI framework [8]. Among other available DBI frameworks, PIN works well in Windows OS and it supports a rich set of APIs for manipulating instrumentation as well as getting dynamic contexts information during program execution.

Our tool works with instruction level granularity: we added analysis functions to instrument memory access activities of every instruction thereby enabling transitions conforming to byte state model.

The shadow memory is attached to the tool (Fig. 3) to populate byte states of every native bytes of monitored process. It is implemented based on a page-table-like structure allowing us to scale memory requirements with the actual process address space in use. To be accorded with byte state model, we adopt 1-byte precision: each byte of native memory is mapped to 2-bit unit in shadow memory, thus, it is enough for our usage (3 states).

To obtain dump of unpacked image (sections corresponding to PE mapped area), our tool involves a module to recognize and handle PE layout. In addition, logging exception and stack unwinding events is also implemented to help OEP finding guesswork which is described in the following subsection.

3.3 Example Runs

Here we describe example runs of our tool to demonstrate how to interpret code exposure measurement with the Windows calculator (`CALC.EXE`, 112 KB). We packed this program with some widely deployed packers: UPX, PECompact, and Yoda Crypter. Figure 4 shows the results of example runs. As shown in Fig. 4 (a), there was no variation detected in the execution of original (non-packed) program as it does not accompany any code exposure behavior. Meanwhile, the other runs exposed hidden code. Observing these graph results (Figs. 4 (b)~4 (d)], each executions exhibited staircase-like behaviors in regards to memory accesses. The flat parts may involve myriads of read and write accesses and the bump increases indicate executions of newly exposed (executed) bytes. Note that the flat parts may also involve executions of already counted (*i.e.*, recurrent executions of same code) or of non-variant/non-SMC code in terms of code exposure (*e.g.*, executions of code already loaded before runtime unpacking).

In many cases, including the cases of UPX and PECompact, measure r is non-decreasing[†] as packed code should be rolled out and then executed (packer principle).

However, we could also observe decreased measure (*i.e.*, *code concealment*) in Yoda Crypter case (Fig. 4 (d)). To verify this code concealing behavior, we manually performed debugging. As a result, we could find out the following behavior in order:

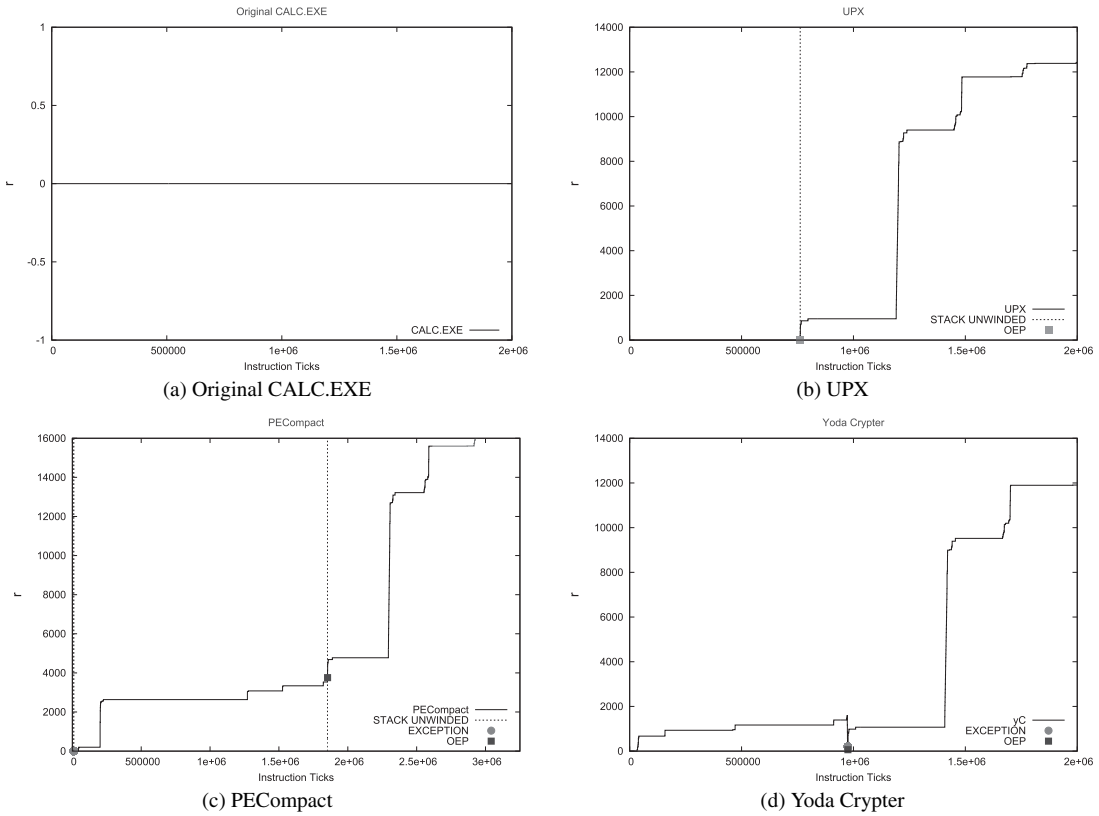
(i) A memory range [0x101f0c6~0x101f994] is filled with some values.
(ii) The above range is executed as code (*i.e.*, the code exposed).
(iii) The above range is filled with zero values (*i.e.*, the code concealed).
(iv) The program control is at OEP.

Observing the executed code in the step (ii), this actually unpacked main code, thus, it is deemed to be a runtime packer (main code loader). This explains that some code acted as a runtime unpacker is generated dynamically and removed later before the control is transfered to OEP.

On the OEP Guesswork. Usually deciding the end of unpacking is proved to be undecidable [16]. Nevertheless, with given code measurement results, it will be helpful to perform guesswork to spot candidate OEPs. Our conjecture, in accordance with packer principle, is that original main code usually starts (at OEP) triggering myriads of state transitions (from $[W]$ to $[W \rightarrow E]$) resulting in abrupt increase of code measure r. Therefore, to collect candidate OEPs, generally it is enough to concentrate area near the bottoms of each abrupt slope in given code measurement results (Figs. 4 (b)~4 (d)).

The best practice dealing with code measurement graph may include other conventional heuristics such as finding points where *stack unwinding* or *exception* event occurs. As our tool supports logging such events, we could superimposed the logged events to the resulted graphs. UPX case (Fig. 4 (b)) is straightforward as we could observe stack unwinding events near its OEP. The binary packed with PECompact (Fig. 4 (c)) triggers stack unwinding and exception events at the very early stage, although, the real OEP was near the other stack unwinding triggered later. In Yoda

[†]If we observe code measure in the long run, we conjecture that it would be converged in some point, unless if given program dynamically exposes code infinitely.



(a) Original CALC.EXE (b) UPX

(c) PECompact (d) Yoda Crypter

Fig. 4 Code measuring results in conformity with byte state model: The x-axis is instruction ticks and the y-axis represents code measure *r*.

Crypter case (Fig. 4 (d)), stack unwinding did not occur, instead, an exception had been triggered before control was reached at its OEP. Overall, these events near each OEPs were well accorded with the points near the bottom of abrupt slopes. In this way, our method will be helped to do OEP guesswork together with other heuristics thereby possibly narrowing the engineering efforts.

4. Evaluation of the Unpacking Tool

In order to examine efficacy and applicability of our DBI-based unpacking implementation, here we present our empirical evaluation results. First, we show an unpacking evaluation based on synthetically packed (thus known) binaries. Then, we look into unpacking results, performed by automatic batch processing, against wild malware samples to figure out workability. Overall, this section interleaves evaluation methods and its results.

4.1 Experiment I: Evaluation on Synthetically Packed Binaries

To check the efficacy with various packers, we took an approach of differentiating between known non-packed binaries and unpacked binaries. For the evaluation purpose, we apply *edit distance* to calculate code section similarity score of given two binaries as the method is well suited for byte data comparisons. In the following, we define a similarity score calculation formula using edit distance.

Similarity score calculation. Let fcs^x and fcs^y be the respective code sections extracted from given two binaries where we assume that initially both to be not $NULL$. The edit distance[†] between two code sections, $ed(fcs^x, fcs^y)$, is the minimum number of byte insertion and deletion operations to convert fcs^x into fcs^y. Then, the similarity between x and y, $sim(x, y)$ is defined as follow:

$$sim(x, y) = 1 - \frac{ed(fcs^x, fcs^y)}{|fcs^x| + |fcs^y|} \quad (1)$$

The $|fcs^x|$ and $|fcs^y|$ are lengths of fcs^x and fcs^y respectively. Note that $1 - sim(x, y)$ is the *dissimilarity* between x and y.

For known non-packed binaries being considered as original codes, we prepared one "hello world" application (OC_1) and two real malware (OC_2[††] and OC_3[†††]) (Table 1). We manually checked that all the three are not packed in terms of code exposure. We have collected 20 packers (Table 2) from the Internet and performed packing jobs on the three binaries. During the packings, we tried to generate

[†]We deploy Levenshtein distance [34].
[††]MD5: 0e134d81e2187a3e7022b8930c889085.
[†††]MD5: 259bd4099f06ab6f153b5b7699034eaf.

Table 1 Original code samples.

Original code	Description	File size (byte)
OC_1	Synthetic "hello world" program	53,248
OC_2	W32.Rbot	232,488
OC_3	W32.Spybot.Gen	258,080

Table 2 Variation number assignment for tested packers.

Packer			Variation
No	Name	Version	Number
1	ASPack	2.2	1 ~ 4
2	ASProtect	1.5 demo	5 ~ 13
3	EXECryptor	2.3.9	14 ~ 21
4	FSG	2.0	22
5	MEW11	SE v1.2	23 ~ 28
6	NeoLite	2.0 trial	29 ~ 36
7	NsPack	2.3	37 ~ 46
8	PackMan	1	47 ~ 55
9	PECompact	3.00.2 trial	56 ~ 67
10	PEDiminisher	0.1	68 ~ 71
11	PEPack	1.0	72 ~ 73
12	PESpin	1.32	74 ~ 84
13	petite	2.3	85 ~ 87
14	telock	0.98	88 ~ 93
15	Themida	2.0.4.0 demo	94 ~ 108
16	UPX	3.03w	109 ~ 115
17	WinUpack	0.39 final	116 ~ 123
18	WWPack	1.20.3.236 demo	124 ~ 130
19	Yoda's Protector	1.03.3	131 ~ 142
20	Yoda's Crypter	1.3	143 ~ 149

variations for each packer by applying available packer options[†] and the resulted packed binary variations are shown in Table 2. We verified the variations ascertaining that all they have different MD5 signatures. Note that not all 149 packings were successful; 137, 140 and 133 packed variations were obtained for each OC_i.

After unpacking the variations with our tool, we extracted code sections from the unpacked (dump) files (using Algorithm 1; see Appendix A for more details). Then, we applied the formula (1) to obtain similarity scores between unpacked codes and the corresponding original codes (OC_i). In case of failure for obtaining dump files, it is naturally regarded that the scores are deemed to be zero. Figure 5 exhibits the resulted similarity calculations. In many cases, we could confirm that the scores is almost 1 meaning that an unpacked code from a variation is identical with its original code. Meanwhile, we also observed that some cases (*e.g.*, PESpin, telock, Themida, EXECryptor, and ASProtect) show low similarity scores. Among them, ASProtect is unpacking-resistant when its *checksum protection* option is enabled. Binaries packed with PESpin and telock seem that these are not compatible with underlying DBI framework as we even could not execute them with basic instruction counting PIN tool.

We also performed a clustering test to observe whether the unpacked binaries, if obtainable, form clusters based on their original codes. For this purpose, we first calculated dissimilarity scores among all the extracted codes of variations, and then grouped clusters with the dissimilarity threshold

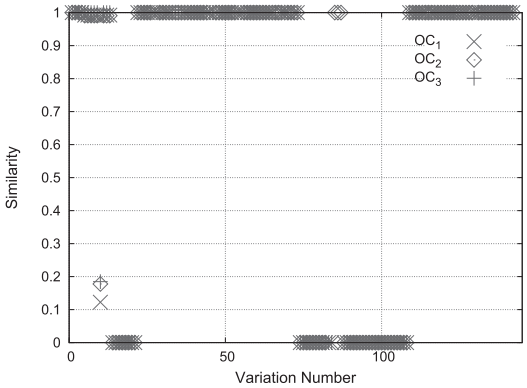


Fig. 5 Similarities between unpacked variations and their corresponding original codes: The x-axis is number assigned for each packed variations and the y-axis is similarity score.

Table 3 Clustering result among the unpacked binaries for each OC_i with $T_d = 0.1$.

Cluster no.	OC_1	OC_2	OC_3
1	89 (98.8%)	0 (0%)	0 (0%)
2	0 (0%)	94 (98.9%)	0 (0%)
3	0 (0%)	0 (0%)	85 (98.8%)
4	1 (0.01%)	1 (0.01%)	1 (0.01%)
Sum	90	95	86

$T_d = 0.1$. Table 3 shows the result of clustering: unpacked variations are grouped into 4 clusters. The clusters 1 to 3 are well accorded with the original codes OC_1 to OC_3 thereby confirming the validity of dump file clustering. There is an exceptional cluster, no. 4, in where all codes were from unpacked binaries originally packed with ASProtect under checksum option. Note that unobtainable dump files are not reflected in the clustering result.

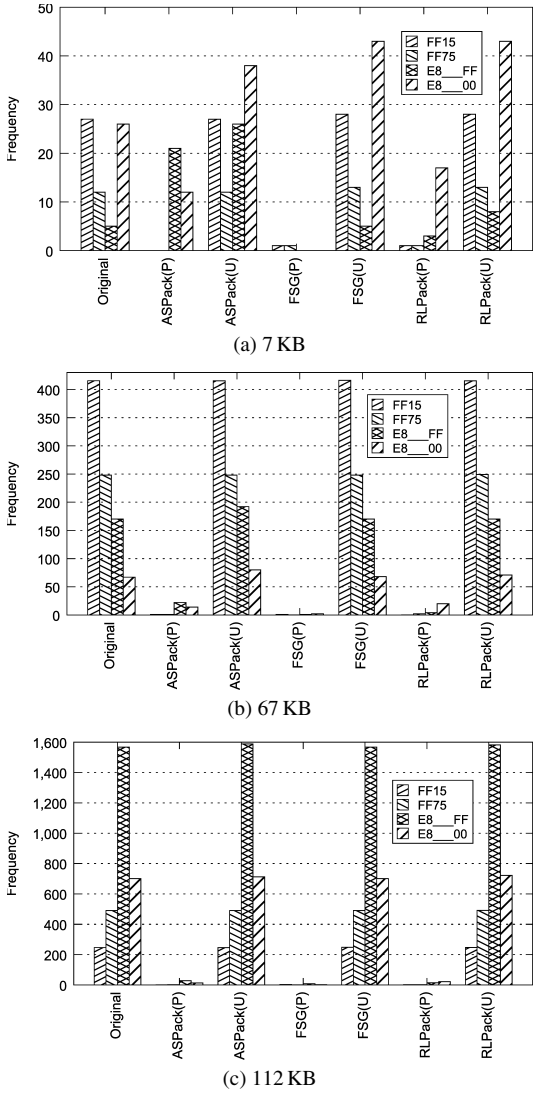
For the reference, we provide an example clustering experiment to figure out how unpacking is effective for malware clustering in Appendix B.

4.2 Experiment II: Bigram Calculations on Malware Samples

In contrast to the last subsection, here we present experimental results with a rather large number of wild malware specimens without the specific prior knowledge of the samples. To evaluate under such assumption, it is plausible to deploy bigram calculation method: *i.e.*, counting frequencies of prevalent `x86` code patterns. In accordance with an prevalency analysis of Sharif et al. [23], we deployed 4 bigrams: `FF 15` (`call`), `FF 75` (`push`), `E8 _ _ _ 00` (`call`), and `E8 _ _ _ FF` (`call`). Notice that the last two bigrams are spaced meaning that any 3 bytes can be placed between both extreme bytes.

To confirm how this method is effective, we first packed known binaries varying sizes (7 KB, 64 KB, and 112 KB) with 3 packers, and then performed unpacking manually

[†]Packers usually provide options such as compression qualities, transformation methods, anti-debugging facilities, and so on.



(a) 7 KB

(b) 67 KB

(c) 112 KB

Fig. 6 Bigram frequencies for packed and unpacked binaries of originally 7 KB, 67 KB, and 112 KB. (P) and (U) mean packed and unpacked respectively. The frequencies of original binaries are at the leftmost.

verifying the unpacked binaries. We calculated bigrams on both the packed and the unpacked binaries. The result is shown in Fig. 6: it is evident that the packed binaries show very low frequencies while the unpacked binaries have relatively high frequencies. As many malware binaries are at around 10 KB–150 KB in its sizes[†], this method would work for usual malware sizes as well as binaries with more larger sizes. Depending the peculiarity of each binaries, resulted patterns were different, however, it was very effective to observe unpacked binaries rooted from packers which exhibit code exposing behavior.

The following two case studies describe test results with real wild malware. As malware emergence may be highly period sensitive, and also we consider to investigate magnified as well as grouped aspects, we separated experiments randomly choosing a one-day and certain periods collections from our malware repository. To deal with a large number of samples, we realized a *batch unpacking processing* taking advantage of the snapshot/revert functions of VMware Workstation. Our unpacking tool was installed in the guest OS (Windows XP SP2). For the termination condition, we set our tool to stop unpacking with the conditions: 1) process termination, 2) 1 minute timeout, and 3) two hundred million instruction execution excess.

Case study (i): one-day collection. We inserted 38 specimens – collected in March 25, 2010 – into the batch system. Among them, we could obtain 32 results (including valid image dump and log files). We tried again those 6 failed samples with a simple instruction counting tool, but PIN could not execute them properly. Among the failed samples, 2 were even not activated appropriately in the guest OS.

Figures from 7 (a) to 7 (d) depict the bigram frequencies for the 32 samples which produced valid dump files. The frequency of an original malware sample and that of unpacked are grouped to be compared with each other. In the results, most samples well expose their hidden code. On the other hand, original and unpacked frequencies of some samples (5, 10, 11, 18, 25, and 29) have no difference. With manual investigations, we concluded that those original samples do not expose hidden code during their executions.

Additionally, by observing 4 bigram patterns associatively, we discovered that same malware were appeared in the same day within the collection. For example, the malware 8, 19, and 24 in Fig. 7 show very similar frequency patterns for all 4 bigrams. With applying an AV scanner, it turned out that the samples were actually same malware. We could bind others resulting in some more groups; Table 4 shows the result. We think that the grouping is possible because the deployed bigrams (mostly `call` instructions) highly reflect program control structures[††].

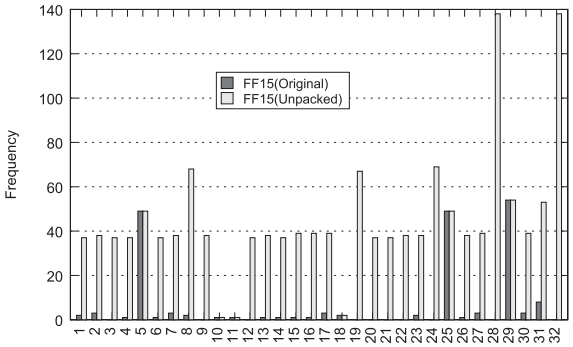
Case study (ii): collections of spanned periods. We have performed unpacking batch processing for two spanned collections: 2750 and 3279 samples were collected within April 1 to 4, 2009 and January 1 to 15, 2010, respectively. Among the all (6029) samples, 12 samples were not valid PE files. Our batch system has produced 1707 and 2147 valid PE dump files for each collections. It took approximately 1 week to process whole samples.

As it is somewhat difficult to present all the individual tendencies for such large set of samples, we used scaled delta (Δ) to reflect relative code frequency difference between original and unpacked binaries: denoting that O_{sum} and U_{sum} are the respective sum of 4 bigram frequencies of original and unpacked binaries, the scaled delta is defined as $\Delta = (U_{sum} - O_{sum})/U_{sum}$. As a binary dynamically generates more new code, its scaled Δ will approach towards 1.

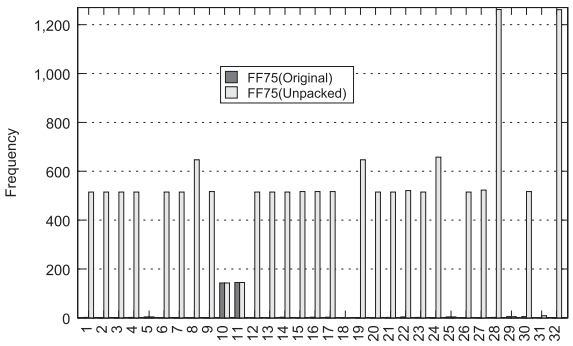
Figures 8 (a) and 8 (b) show the results for each collec-

[†]According to Fortinet collection [35], the sizes of 50% of malware binaries are between 10 KB–100 KB.

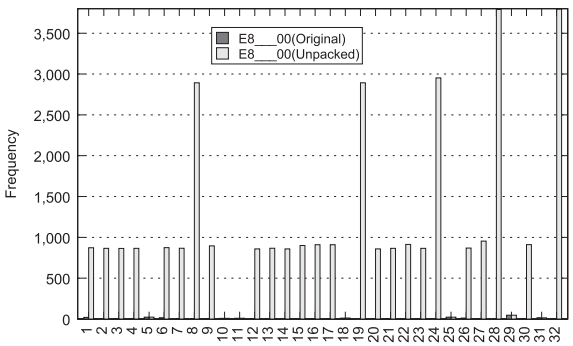
[††]Even though static signatures are different, it is usual that similar malware binaries share similar control flow structure.



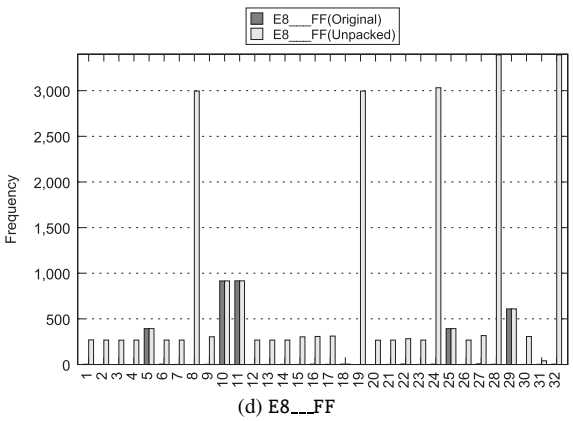
(a) FF15



(b) FF75



(c) E8___00



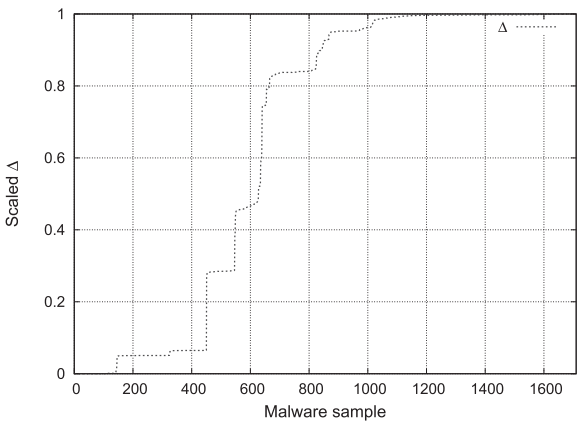
(d) E8___FF

Fig. 7 Bigram frequencies obtained from 32 malware specimens.

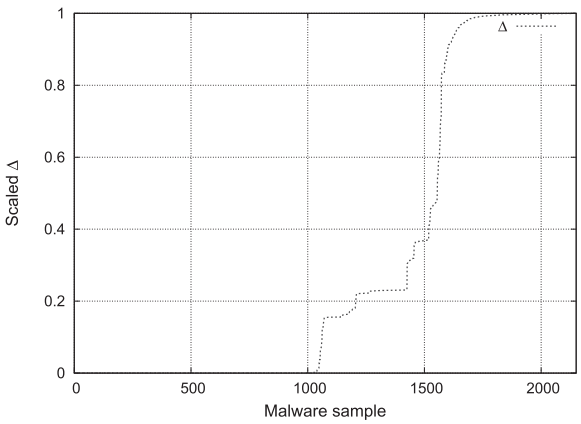
tions. Among each unpacked group originated from each collection, we can observe that, respectively, at around 94 and 53 percents exhibited code exposure behaviors. The difference between the two distributions confirmed us that

Table 4 AV (Symantec) scan result of 32 samples appeared in Fig. 7.

Sample no.	Scan result
1, 2, 3, 4, 6, 7, 9, 12, 13, 15, 16, 17, 20, 21, 22, 23, 26, 27, 30	w32.rahack.w
5, 25, 29	w32.ircbot, backdoor.irc.bot
8, 19, 24	w32.rahack.h
10, 11	backdoor.irc.bot, trojan horese
18	w32.ircbot
28, 32	trojan horse
31	trojan horse
14	NULL



(a) Apr. 1 – 4, 2009



(b) Jan. 1 – 15, 2010

Fig. 8 The distributions of bigram frequency delta for two spanned collections. The delta values are sorted for the sake of distinctness.

the spanned malware collections are period sensitive: many similar malware can be grouped within each collection and the malware emergence tendencies would be different between the collections.

To inquire into approximate workability in native systems (*i.e.*, without any virtualization layers and monitoring facilities), the two collections were also inserted into our other analysis system comprised of native machines. From this test, we confirmed that 660 samples were not activated appropriately.

5. Discussion

In this section, we discuss the applicability of DBI based unpacking tool in the light of the empirical experiences described in the previous section.

With building a fully automatic batch system, without any manual interventions, we could obtain at around 64% of valid PE dump files from the two spanned collections (6029 samples). Excluding corrupted PEs and not well activated samples in native systems, it was approximately 72%. Although we could not delve into all the individuals, the aspects about the unobtainable specimens may involve following considerations.

Workability with DBI. Some malware could not be executed under PIN. In the packer test, our unpacking tool as well as basic instruction counting tool cannot be executed appropriately for 5 out of 20 packers. Compared to 64% (overall) or 72% (purging unworkable samples) workability of case study (ii), Reynaud reported another result [31] deploying a PIN tool for malware analysis (though not for unpacking): 48404 out of 62498 Nepenthes [36] samples were able to analyze (77%). Note that the workability would highly depend on the ability of involved malware collecting systems as well as collection periods.

Configuration of batch processing. To deal with a large number of samples, it is inevitable to build an automatic batch system. The deployed configuration as well as underlying base systems would influence on the efficacy. As mentioned in Sect. 4.2, we applied some terminal conditions to finalize recurrent unpacking sessions. For example, timeout condition might be eluded by or not accorded with some malware which are deliberately inactive within a couple of minutes.

Our reference batch system is built on VMware for environmental restoration and isolation. Even though VMware Workstation can mitigate some VM detection issues by appropriately configuring some options [37], new detection methods, *e.g.* [38], has been emerging as arms races. In addition, currently network emulation unit is not attached to our system. This also may be a factor in drawing more code exposing activities from malware.

Collection peculiarities. Because malware emergence patterns are hard to expect and some malware may be prominent in a specific period span, it is evident that workability results would be floating among collections of different period spans.

Among the analyzable samples, as described in Sect. 4.2, the unpacking observation in terms of code exposure is different. Some samples, which does not expose hidden code, may be actually packed in the other way: *e.g.*, protected by virtualization component [39] embedded within binaries. Such behavior cannot be detected by unpacker tools depending and enforcing common unpacking heuristic which mostly expect to detect dynamic new code generation.

Overall, considering the above conjectures, we think that there are rooms for fidelity enhancement as well as configuration improvement. Active manual involvement, for dealing not general but specific targets, may also increase workability. In the past, we also have some experiences with the other platforms and the following is the note about it.

Empirical framework experiences. As we explored in related work, there are other available frameworks to be deployed. Each have pros and cons over each other. Because our required function is to instrument processes in fine-grained manner, first we have tested some samples in two plausible platforms, PIN DBI and Ether/Xen. Ether provides very fast and transparent malware analysis environment as the underlying Xen is based on hardware supported virtualization. However, it seems rather slow and yet unstable when fine-grained tracing is enabled [40]. The slowness of the fine-grained mode, as noted in Sect. 2.1, seems to be incurred by trap mechanism which involve VM context changes for every tracked instructions. We also had built some analysis modules on Bochs [41] and QEMU [42]. Bochs is better support in building plug-ins as well as working with fine-grained manipulation; meanwhile, QEMU was substantially faster than Bochs. As Bochs, QEMU, and Xen are based on whole system virtualization, they can provide inherent environmental isolation while performing instrumentation, and contrarily PIN requires additional encapsulation for malware analysis. However, PIN supports a rich set of well-defined APIs for fine-grained instrumentation; thus, tool development for fine-grained analysis is more facilitated. In the other platform cases, the understanding of underlying virtualization architecture – sometimes involving core engine source codes – is required: usually, it is not easy to get into from the first time.

During the research involvement, we witnessed that framework enhancements can lead to the improvement of analysis ability. For example, when we first built our prototype in PIN [43], Yoda Crypter did not work well with PIN while with the recent version enables it to be executed. Similarly, the recent version of QEMU seems to work with telock packer while older versions did not.

6. Conclusion

This paper explored the applicability of a DBI-based unpacking tool based on our empirical experimental experiences. First, we presented a target unpacking method implemented on PIN DBI framework. The core design of our tool is to measure code exposure in accordance with byte state model. Then, we described experimental results of batched unpacking processing performed on real wild malware specimens (one-day and spanned collections). As for case studies of a DBI-based tool implementation for dynamic analy-

sis, we believe that our experimental data presented in this work will be helpful in deployment decisions of malware analysis systems.

Currently, we are working on building automatable malware analysis system for primary binary analysis on QEMU whole system emulator. In parallel, we plan to develop a method to mitigate memory checksum problem as well as other measures to enhance analysis quality of DBI-based tools. Our interests also include dealing with binaries protected with virtualization techniques.

Acknowledgments

We would like to thank Jumpei Shimamura and Yaichiro Takagi for their technical assistance.

References

[1] W. Yan, Z. Zhang, and N. Ansari, "Revealing packed malware," IEEE Security and Privacy, vol.6, no.5, pp.65–69, 2008.
[2] T. Brosch and M. Morgenstern, "Runtime packers: The hidden problem?," Black Hat USA, 2006.
[3] VMWare. http://www.vmware.com/
[4] 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. Nineteenth ACM Symposium on Operating Systems Principles (SOSP), pp.164–177, ACM, New York, NY, USA, 2003.
[5] KVM. http://www.linux-kvm.org/
[6] D. Mihocka and S. Shwartsman, "Virtualization without direct execution or jitting: Designing a portable virtual machine infrastructure," 1st Workshop on Architectural and MicroArchitectural Support for Binary Translation (AMAS-BT), 2008.
[7] F. Bellard, "Qemu, a fast and portable dynamic translator," Proc. Annual Conference on USENIX Annual Technical Conference (ATEC), pp.41–46, USENIX Association, Berkeley, CA, USA, 2005.
[8] 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," Proc. 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp.190–200, ACM, New York, NY, USA, 2005.
[9] N. Nethercote and J. Seward, "Valgrind: A framework for heavy-weight dynamic binary instrumentation," Proc. 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp.89–100, ACM, New York, NY, USA, 2007.
[10] D. Bruening, Efficient, Transparent, and Comprehensive Runtime Code Manipulation, Ph.D. thesis, Massachusetts Institute of Technology, Sept. 2004.
[11] C. Wang, S. Hu, H.-S. Kim, S.R. Nair, M. Breternitz, Jr., Z. Ying, and Y. Wu, "Stardbt: An efficient multi-platform dynamic binary translation system," Advances in Computer Systems Architecture, Lect. Notes Comput. Sci., vol.4697, pp.4–15, Springer, 2007.
[12] B. Buck and J.K. Hollingsworth, "An API for runtime code patching," Int. J. High Perform. Comput. Appl., vol.14, no.4, pp.317–329, 2000.
[13] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," Proc. 15th ACM Conference on Computer and Communications Security (CCS), pp.51–62, ACM, New York, NY, USA, Oct. 2008.
[14] P. Royal, "Alternative medicine: The malware analyst's blue pill," Black Hat USA, 2008.
[15] M.G. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code extractor for packed executables," Proc. 2007 ACM Workshop on Recurring Malcode (WORM), pp.46–53, ACM, New York, NY, USA, 2007.
[16] 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), pp.289–300, IEEE Computer Society, Washington, DC, USA, 2006.
[17] L. Böhne, Pandora's bochs: Automatic unpacking of malware, Master's thesis, RWTH Aachen University, 2008.
[18] D. Quist and Valsmith, "Covert debugging: Circumventing software armoring techniques," Black Hat USA, 2007.
[19] J.Y. Marion and D. Reynaud, "Dynamic binary instrumentation for deobfuscation and unpacking," In-Depth Security Conference 2009 (DeepSec), Nov. 2009. http://code.google.com/p/tartetatintools/
[20] P. Bania, "Generic unpacking of self-modifying, aggressive, packed binary programs," http://piotrbania.com/all/articles/pbania-dbi-unpacking2009.pdf, 2009.
[21] N. Nethercote, "Dynamic binary analysis and instrumentation," Tech. Rep. UCAM-CL-TR-606, 2004.
[22] L. Sun, T. Ebringer, and S. Boztas, "Hump-and-dump: Efficient generic unpacking using an ordered address execution histogram," 2nd International Computer Anti-Virus Researchers Organization (CARO) Workshop, 2008.
[23] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee, "Eureka: A framework for enabling static malware analysis," ESORICS, Lect. Notes Comput. Sci., vol.5283, pp.481–500, Springer, 2008.
[24] J. Stewart, "Semi-automatic unpacking on IA-32 using ollybone," RECON, 2006.
[25] L. Martignoni, M. Christodorescu, and S. Jha, "Omniunpack: Fast, generic, and safe unpacking of malware," Proc. 23rd Annual Computer Security Applications Conference (ACSAC), pp.431–441, 2007.
[26] F. Guo, P. Ferrie, and T. Chiueh, "A study of the packer problem and its solutions," Recent Advances in Intrusion Detection (RAID), Lect. Notes Comput. Sci., vol.5230, pp.98–115, Springer, 2008.
[27] S. Josse, "Secure and advanced unpacking using computer emulation," J. Computer Virology, vol.3, no.3, pp.221–236, 2007.
[28] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M.G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," ICISS, Lect. Notes Comput. Sci., vol.5352, pp.1–25, Springer, 2008.
[29] M. Christodorescu, S. Jha, J. Kinder, S. Katzenbeisser, and H. Veith, "Software transformations to improve malware detection," J. Computer Virology, vol.3, no.4, pp.253–265, 2007.
[30] K.A. Roundy and B.P. Miller, "Hybrid analysis and control of malware binaries," http://www.paradyn.org/, 2009.
[31] D. Reynaud, "A look at anti-virtualization in malware samples," 2009. http://indefinitestudies.org/
[32] OpenBSD. http://www.openbsd.org/
[33] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," Proc. 3rd International Conference on Virtual Execution Environments (VEE), pp.65–74, ACM, New York, NY, USA, 2007.
[34] V.I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," Soviet Physics Doklady, vol.10, no.8, pp.707–710, 1966.
[35] B. Lu, "A deeper look at malware — the whole story," Virus Bulletin Conference (VB), 2007.
[36] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling, "The nepenthes platform: An efficient approach to collect malware," Recent Advances in Intrusion Detection (RAID), Lect. Notes Comput. Sci., vol.4219, pp.165–184, Springer, 2006.
[37] T. Liston and E. Skoudis, "On the cutting edge: Thwarting virtual machine detection," SANSFIRE 2006, 2006.
[38] "Scoopyng — the vmware detection tool," http://www.trapkit.de/research/vmm/scoopyng/index.html
[39] VMProtect. http://www.vmprotect.ru/

[40] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna, "Efficient detection of split personalities in malware," Proc. 17th Annual Network and Distributed System Security Symposium (NDSS), 2010.
[41] H.C. Kim, D. Inoue, M. Eto, J. Song, and K. Nakao, "An implementation of a generic unpacking method on bochs emulator," CSS 2009: The 2009 Computer Security Symposium, pp.997–1002, 2009.
[42] H.C. Kim, D. Inoue, M. Eto, J. Song, and K. Nakao, "An extended qemu emulator for primary binary analysis," SCIS 2010: The 2010 Symposium on Cryptography and Information Security, IEICE, 2010.
[43] H.C. Kim, D. Inoue, M. Eto, T. Yaichiro, and K. Nakao, "Toward generic unpacking techniques for malware analysis with quantification of code revelation," Joint Workshop on Information Security, Aug. 2009.
[44] Kaspersky. http://www.kaspersky.com/
[45] McAfee. http://www.mcafee.com/

Appendix A: Extracting Code Section

It might be possible to take whole dumped binaries to be treated as targets for similarity score calculations. As our tool depends on code exposure, we aimed to only deal with code sections: dumped binaries by unpacking may involve other parts which might be considered as noise incurred by dynamic situations per runs. Therefore, for experiments I and III in Sect. 4, we try to extract code sections from dumped files exploiting that a code section contains prevalent `x86` instruction byte patterns such as `push` and `call` according with bigram method described in Sect. 4.2. The algorithm 1 is devised for the purpose: it scans a given binary, with sliding window approach, and extracts candidate code sections if encountering byte range oc contains code patterns beyond defined code appearance rate T_i. Through the evaluation experiments, we use $w = 2048$ (window size), $T_{len} = 512$, and $T_i = 0.01$. To choose these parameters, we investigated benign 350 executables (originally resided in Windows XP) with applying several combinations. As shown in Fig. A· 1, we could identify that we may be able to discriminate code sections from others with the selection.

Algorithm 1 `code_ext`

```
function code_ext(oc, w, T_i, T_len) {
    if (|oc| < T_len) return NULL;
    for (s = 0; s < |oc| − w; s++ ) {
        if (machine_code(oc[s, s + w − 1]) > T_i) {
            for (start = s; mc(oc[start, start]) == 0; start++) ;
            break;
        }
    }
    if (s == |oc| − w) return NULL;
    for (e = |oc| − 1; e >= 0; e++ ) {
        if (machine_code(oc[e − w + 1, e]) > T_i) {
            for (end = e; mc(oc[end, end]) == 0; end++) ;
            break;
        }
    }
    if (end − start + 1 > T_len) return oc[start, end];
    else return NULL;
}
```

Appendix B: A Malware Clustering Example

This section presents an example clustering test performed on 236 real malware samples. The malware samples we used were collected, between June and July of 2008, using a well-known low interaction honeypot system Nepenthes [36].

We conducted a clustering experiment with the same way as in Sect. 4.1: Table A· 1 shows the results obtained by varying the dissimilarity threshold values T_d from 0 to 0.5. The malware were grouped into 20~56 clusters depending on the thresholds. For the reference, we detailed the clustering result of $T_i = 0.1$ affixing malware names obtained by applying two commercial antivirus software products (Kaspersky [44] and McAfee [45]), and that was shown in Table A· 2. From the clutering result, we could confirm a certain degree of consentaneity among the grouped samples.

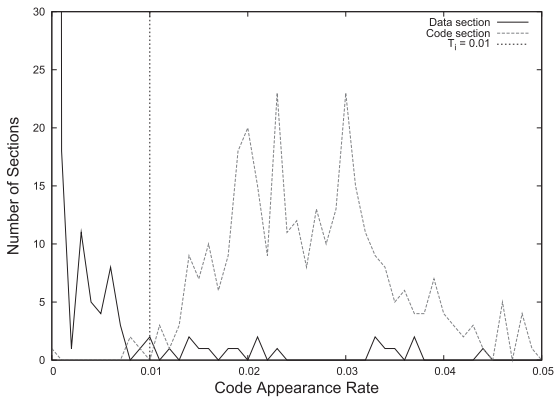


Fig. A· 1 Parameter selection for Algorithm 1. With the window size $w = 2048$, code sections are well reacted over $T_i = 0.01$ while data sections are not.

Table A· 1 Number of resulted clusters.

Threshold	Number of clusters
$T_d = 0$	56
$T_d = 0.01$	46
$T_d = 0.05$	31
$T_d = 0.10$	26
$T_d = 0.50$	20

Table A· 2 Clustering result of 236 malware samples with $T_d = 0.1$.

Cluster	Kaspersky	Count	McAfee	Count
1	Backdoor.Win32.Agent	1	W32/Virut	7
	Backdoor.Win32.Rbot	1	W32/Sdbot.worm	2
	Backdoor.Win32.VanBot	2	Generic	1
	Virus.Win32.Virut	8	Generic BackDoor	1
			W32/Bobax.worm	1
2	Net-Worm.Win32.Bobic	1	W32/Bobax.worm	1
3	Virus.Win32.Virut	10	W32/Virut.gen	11
	Backdoor.Win32.VanBot	3	Generic BackDoor	3
	Net-Worm.Win32.Kolabc	2	W32/Bobax.worm	1
4	Backdoor.Win32.VanBot	6	W32/Virut	5
	Virus.Win32.Virut	3	W32/Bobax.worm	3
	Net-Worm.Win32.Kolabc	2	Generic	2
	Net-Worm.Win32.Bobic	1	Generic BackDoor	1
			W32/Sdbot.worm	1
5	Backdoor.Win32.VanBot	1	Generic BackDoor	1
6	Virus.Win32.Virut	8	W32/Bobax.worm	8
	Net-Worm.Win32.Bobic	3	W32/Virut	3
	Backdoor.Win32.IRCBot	1	W32/Sdbot	1
7	Virus.Win32.Virut	1	W32/Virut	1
8	Trojan-Dropper.Win32.Sramler	13	W32/Virut	9
			Generic	2
			W32/Bobax.worm	2
9	Backdoor.Win32.Agent	1	W32/Sdbot.worm	1
10	Virus.Win32.Virut	6	W32/Virut	6
11	Virus.Win32.Virut	2	W32/Virut	2
12	Trojan-GameThief.Win32.OnLineGames	1	Generic BackDoor	1
13	Trojan-Dropper.Win32.Sramler	1	W32/Virut	1
14	Backdoor.Win32.VanBot	1	W32/Sdbot.worm	1
15	Virus.Win32.Virut	1	W32/Virut	1
16	Backdoor.Win32.Nepoe	19	W32/Virut	34
	Net-Worm.Win32.Kolabc	18	W32/Bobax.worm	13
	Backdoor.Win32.VanBot	11	Generic BackDoor	7
	Virus.Win32.Virut	8	Generic	3
	Net-Worm.Win32.Bobic	3	Generic Malware	1
	Backdoor.Win32.IRCBot	1	New Malware	1
	Trojan-Dropper.Win32.Sramler	1	W32/Nirbot	1
			W32/Sdbot	1
17	Net-Worm.Win32.Kolabc	19	W32/Virut	13
	Virus.Win32.Virut	1	W32/Bobax.worm	5
			Generic BackDoor	2
18	Net-Worm.Win32.Bobic	1	W32/Bobax.worm	1
19	Virus.Win32.Virut	2	W32/Virut.gen	2
20	Virus.Win32.Virut	1	W32/Bobax.worm	1
21	Trojan-Dropper.Win32.Sramler	1	W32/Virut	1
22	Virus.Win32.Virut	4	W32/Bobax.worm	3
	Backdoor.Win32.Rbot	1	W32/Poebot	1
			W32/Virut	1
23	Virus.Win32.Virut	1	W32/Virut	1
24	Backdoor.Win32.Rbot	6	W32/Sdbot.worm	6
25	Backdoor.Win32.EggDrop	1	Generic	1
26	Backdoor.Win32.Rbot	1	Unknown	1
27	Backdoor.Win32.Rbot	1	W32/Sdbot.worm	1
28	Net-Worm.Win32.Padobot	1	W32/Korgo.worm	1
29	Backdoor.Win32.VanBot	1	Generic BackDoor	1
30	Trojan.Win32.Pakes	1	W32/Sdbot.worm	1
31	Virus.Win32.Virut	1	W32/Virut	1
32	Virus.Win32.Virut	8	W32/Virut	7
			Generic	1
33	Virus.Win32.Virut	1	W32/Bobax.worm	2
	Net-Worm.Win32.Bobic	1		
34	Net-Worm.Win32.Bobic	1	W32/Bobax.worm	1
35	Trojan.Win32.Qhost	1	Generic BackDoor	1
	Net-Worm.Win32.Bobic	1	W32/Bobax.worm	1
36	Virus.Win32.Virut	1	W32/Virut	1
37	Backdoor.Win32.Rbot	10	W32/Sdbot.worm	11
	Net-Worm.Win32.Kolabc	1		
38	Virus.Win32.Virut	4	W32/Virut	4
	Net-Worm.Win32.Bobic	3	W32/Bobax.worm	4
	Backdoor.Win32.VanBot	2	Generic BackDoor	1
39	Virus.Win32.Virut	1	W32/Virut	1
40	Net-Worm.Win32.Bobic	1	W32/Bobax.worm	1
41	Virus.Win32.Virut	4	W32/Bobax.worm	2
			Generic BackDoor	1
			W32/Virut	1
42	Trojan.Win32.Pakes	1	W32/Sdbot.worm	1
	Backdoor.Win32.Rbot	1	Exploit-DcomRpc	1
43	Net-Worm.Win32.Bobic	1	W32/Bobax.worm	1
44	Backdoor.Win32.Nepoe	1	W32/Sdbot.worm	1
45	Backdoor.Win32.IRCBot	1	W32/Sdbot.worm	1
46	Backdoor.Win32.VanBot	3	W32/Bobax.worm	3
	Trojan-Dropper.Win32.Agent	2	W32/Virut	2
	Net-Worm.Win32.Bobic	1	W32/Sality	1

Hyung Chan Kim received the B.S. in Computer Science from Kyungpook National University, Daegu, Korea, in 2001 and the M.S. and Ph.D. in Information and Communications from Gwangju Institute of Science and Technology (GIST), Gwangju, Korea in 2003 and 2007, respectively. He was a postdoctoral researcher in Department of Computer Science at Columbia University in the city of New York, NY, USA in 2008. From 2009 to 2010, he was an expert researcher in National Institute of Information and Communications Technology (NICT), Tokyo, Japan. He joined The Attached Institute of ETRI, Daejeon, Korea, in December 2010. His recent research interests include security issues in operating systems, virtualization platforms, and embedded systems. He received the Student Paper Award at the 2005 Computer Security Symposium (CSS 2005).

Tatsunori Orii received his B.E. in Electrical and Computer Engineering from Yokohama National University in 2009. He is currently a master course student at the Graduate School of Environment and Information Sciences, Yokohama National University. His research interest covers a wide area of network security including malware analysis.

Katsunari Yoshioka received the B.E., M.E. and Ph.D. degrees in Computer Engineering from Yokohama National University in 2000, 2002, 2005, respectively. From 2005 to 2007, he was a Researcher at the National Institute of Information and Communications Technology, Japan. Currently, he is an Associate Professor for Division of Social Environment and Informatics, Graduate School of Environment and Information Sciences, Yokohama National University. His research interest covers a wide range of information security, including malware analysis, network monitoring, intrusion detection, and information hiding. He was awarded 2009 Prizes for Science and Technology by The Commendation for Science and Technology by the Minister of Education, Culture, Sports, Science and Technology.

Daisuke Inoue received his B.E. and M.E. degrees in electrical and computer engineering from Yokohama National University in 1998 and 2000, respectively, and Ph.D. degree in engineering from Yokohama National University in 2003. He joined the Communications Research Laboratory (CRL), Japan, in 2003. The CRL was relaunched as the National Institute of Information and Communications Technology (NICT) in 2004, where he is a senior researcher of the Information Security Research Center. His research interests include security and privacy technologies in wired and wireless networks, incident handling based on network monitoring and malware analysis. He received the Best Paper Award at the 2002 Symposium on Cryptography and Information Security (SCIS 2002).

Jungsuk Song received his B.S. and M.S. degrees in Information and Telecommunication Eng. from Korea Aerospace University, Korea in 2003 and 2005, respectively. He received his Ph.D. degree in the Graduate School of Informatics, Kyoto University, Japan in 2009. He is currently an expert researcher at National Institute of Information and Communications Technology (NICT), Japan. His research interests include network security, data mining, machine learning, security issues on IPv6, spam analysis, and cryptography theory. He is a member of IEEE.

Masashi Eto received LL.B degree from Keio University in 1999, received the M.E. and Ph.D. degrees from Nara Institute of Science and Technology (NAIST) in 2003, 2005, respectively. From 1999 to 2003, he was a system engineer at Nihon Unisys, Ltd., Japan. He is currently a researcher at National Institute of Information and Communications Technology (NICT), Japan. His research interests include network monitoring, intrusion detection, malware analysis and auto-configuration of the Internetworking. He received the Best Paper Award at the 2007 Symposium on Cryptography and Information Security (SCIS 2007).

Junji Shikata received the B.S. and M.S. degrees in mathematics from Kyoto University, Kyoto, Japan, in 1994 and 1997, respectively, and the Ph.D. degree in mathematics from Osaka University, Osaka, Japan, in 2000. From 2000 to 2002 he was a Postdoctoral Fellow at the Institute of Industrial Science, the University of Tokyo, Tokyo, Japan. Since 2002 he has been with the Graduate School of Environment and Information Sciences, Yokohama National University, Yokohama, Japan. From 2008 to 2009, he was a visiting researcher at the Department of Computer Science, Swiss Federal Institute of Technology (ETH), Zurich, Switzerland. Currently, he is an Associate Professor of Yokohama National University. His research interests include cryptography, theoretical computer science and computational number theory. Dr. Shikata received several awards including the TELECOM System Technology Award from the Telecommunications Advancement Foundation in 2004, the Wilkes Award 2006 from the British Computer Society, and the Young Scientists Prize, the Commendation for Science and Technology by the Minister of Education, Culture, Sports, Science and Technology in Japan in 2010.

Tsutomu Matsumoto is a Professor of Division of Social Environment and Informatics, Graduate School of Environment and Information Sciences, Yokohama National University. His current roles include an Associate Member of the Science Council of Japan, an Advisor of Research Center for Information Security, National Institute of Advanced Industrial Science and Technology, and a core member of CRYPTREC — the Cryptography Research and Evaluation Committees for governmental use of cryptographic technology. Starting from design and analysis of various cryptosystems and cryptographic protocols in the early 80s, he has opened up the field of security measuring for logical and physical security mechanisms including human-machine cryptography, information hiding, software security, biometric security, side-channel security, and artifactmetrics. He got Doctor of Engineering degree from the University of Tokyo in 1986. He received the Achievement Award from IEICE in 1995, the DoCoMo Mobile Science Award in 2006, the Culture of Security Award in 2008, and the Prize for Science and Technology, the commendation by the Minister of Education, Culture, Sports, Science and Technology in 2010.

Koji Nakao is the Information Security Fellow in KDDI, Japan. Since joining KDDI in 1979, he has been engaged in the research on multimedia communications, communication protocol, secure communicating system and information security technology for the telecommunications network. He is also an active member of Japan ISMS user group, which was established in the 1st Quarter of 2004. He is the board member of Japan Information Security Audit Association (JASA) and that of Telecom-ISAC Japan, and concurrently, a Technical Group Chairs (ICSS: information communication system security) of the Institute of Electronics, Information and Communication Engineers. He received the B.E. degree of Mathematics from Waseda University, in Japan, in 1979. He received the IPSJ Research Award in 1992, METI Ministry Award and KPMG Security Award in 2006, Contribution Award (Japan ITU), NICT Research Award, Best Paper Award (JWIS) and MIC Bureau Award in 2007. He is a member of IPSJ. He has also been a part-time instructor in Waseda University since 2002.