SUMMARY Many malware programs emerging from the Internet are compressed and/or encrypted by a wide variety of packers to deter code analysis, thus making it necessary to perform unpacking first. To do this task efficiently, Guo et al. proposed a generic unpacking system named Justin that provides original entry point (OEP) candidates. Justin executes a packed program, and then it extracts written-and-executed points caused by the decryption of the original binary until it determines the OEP has appeared, taking those points as candidates. However, for several types of packers, the system can provide comparatively large sets of candidates or fail to capture the OEP. For more effective generic unpacking, this paper presents a novel OEP detection method featuring two mechanisms. One identifies the decrypting routine by tracking relations between writing instructions and written areas. This is based on the fact that the decrypting routine is the generator for the original binary. In case our method fails to detect the OEP, the other mechanism sorts candidates based on the most likely candidate so that analysts can reach the correct one quickly. With experiments using a dataset of 753 samples packed by 25 packers, we confirm that our method can be more effective than Justin’s heuristics, in terms of detecting OEPs and reducing candidates. After that, we also propose a method combining our method with one of Justin’s heuristics.

key words: software packer, malware, code analysis, Data Execution Prevention (DEP), security

1. Introduction
Malicious software (malware) poses many pressing challenges for malware analysts, in particular the need to defeat obfuscation techniques. The techniques can produce the capability of shielding malware from code analysis (also known as static analysis [1]). Even worse, malware authors can very easily compress and/or encrypt the original binaries by using software packers [2] (e.g., UPX [3], ASPack [4], PECompact [5], and Themida [6]). Analysts have to unwrap packed malware samples, as a usual way, by inspecting every packing algorithm used for the malware. They spend much time on performing manual unpacking, while many new packers and versions are developed constantly.

One way analysts have studied to solve this packing problem is generic unpacking. It is defined as a type of methods that automatically perform unpacking without depending on packer types used for malware. To provide helpful information for code analysis, generic unpacking methods [7]–[13] focus on elementary steps possessed by packed programs: after a packed program is loaded into the memory, the original binary is first decrypted and written by the relevant unpacking (decrypting) routine, and then the instruction pointer transfers to the original binary. Generally, that beginning point called original entry point (OEP) is considered very helpful. That is, if an analyst can recognize the OEP, she can easily analyze the unpacked original binary as the way of debugging [14]. Thus methods for detecting the OEP are strongly required as a kind of generic unpacking.

There exist two basic techniques applied to the detection of the OEP: single-step execution and written-and-executed memory page extraction [15], where memory page means a fixed-size area on the process memory. The single-step execution is a technique that can monitor and disassemble each instruction (e.g., ‘push eax’ and ‘mov eax, ebx’) of a running process; however, it induces a significant performance overhead [2], [13]. The other technique is to only extract written-and-executed memory pages and instructions that wrote some data to those pages. It has the advantage of very low performance overhead, whereas it just provides such limited resources for unpacking. Both techniques have their own advantages, but the page extraction technique is suitable for our purpose. That is, by providing the OEP quickly, we aim to cope with the serious situation in which some analysts have to efficiently analyze the sheer volume of malware.

The OEP will be hidden among the set of written-and-executed points because the original binary of packed malware is first decrypted (written) and then executed. We can track such written-and-executed points with the page extraction technique. However, the obtained set of executed points can be large for analysts even if a malware sample is packed by a simple packer such as UPX. Considering those executed points as OEP candidates, we need to filter out false-positive (not-OEP) candidates as many as possible. To do this task, Guo et al. adopt the page extraction technique and devise a generic unpacking system called Justin [13], which possesses some heuristics to filter out the candidates. The most effective one of its heuristics is to check if the stack pointer value is equal to the initial one given at the start of a packed sample’s run. This check is conducted every time a written memory page is executed. If the stack pointer has been initialized, Justin provides the set of candidates that it has obtained until the condition is satisfied. This heuristic is based on the fact that most packers clean up the stack after the decryption of the original binary is finished. Justin, however, can provide comparatively large candidate sets for
several types of packers due to some written points that are executed before the stack pointer is initialized. In addition, it can fail to capture the OEP (i.e., it can provide a candidate set without the correct one) because some packers do not clean up the stack to evade generic unpacking.

In this paper, we adopt the written-and-executed page extraction technique, and we aim at detecting the OEP more effectively and reducing the number of candidates. To this end, we propose a novel OEP detection method featuring two mechanisms. First, it identifies memory pages of the unpacking routine by tracking relations between writing instructions (e.g., ‘mov’ and ‘xchg’) and written memory pages. This is based on the fact that the unpacking routine is a generator for the original binary. The other memory pages can be considered as part of the original binary, and the first-executed point on the areas can be the OEP. Second, our method provides executed points as OEP candidates after sorting the candidates ranked by the OEP detected as the most likely one. In contrast, Justin does not consider the sorting of candidates. The sorted set of candidates can be helpful for analysts in case our method fails to detect the correct one. That is, analysts can quickly get the correct OEP, checking candidates from the top in turn.

The experiments in this paper evaluate our method with a dataset of 753 malware samples packed by 25 packers, in terms of detecting the OEP and reducing candidates. The results show that our method uniquely detected the correct OEP for 81% of all the packed samples. In addition, our method was able to gather the correct OEP within the top six candidates for 92% of all the samples by the sorting mechanism. We also evaluate Justin’s heuristics, comparing our method with them. This evaluation confirms that our method can be more effective than Justin’s heuristics from the perspective of detecting OEPs for the whole dataset. Our method also succeeded in trimming OEP candidates more effectively than Justin’s heuristics for samples packed by several packers. After examining the results, we propose a method combining our method with one of Justin’s heuristics for obtaining better results.

We choose Justin for comparison in this paper although some packers have already tried to evade Justin’s heuristics. This is because, as far as we know, Justin is still the most effective one among unpacking systems that adopt the page extraction technique. Besides, ACM survey papers [2] and [15], which cover unpacking methods, introduce only Justin and OmniUnpack as that type of systems. OmniUnpack is described in Sect. 2.3 in this paper.

Our main contributions in this paper are to track relations between writing instructions and written memory pages by using the page extraction technique; to sort OEP candidates; and to combine our method with one of Justin’s heuristics. The rest of this paper is organized as follows. In Sect. 2, we introduce the background of our research including related work. In Sect. 3, we present our method. In Sect. 4, we evaluate our method and Justin’s heuristics, and then we propose and evaluate a combined method. Finally, we conclude this paper in Sect. 5.

2. Background

2.1 Packer Definition and Model

A packer is a software tool that compresses or encrypts a program, and attaches its unpacking routine to the packed binary. This routine can give a self-executable form to the packed program. As an example, Fig. 1 shows the file structure of a program packed by UPX [13]. It mainly consists of a portable executable header (PE header), an empty section, the packed binary, and the relevant unpacking routine of UPX. When this program is executed, the unpacking routine first decrypts the packed binary, while writing the decrypted original code and data (i.e., pieces of the original binary) to the empty section. Then the instruction pointer moves to the beginning address on the original binary called original entry point (OEP). After that, the original binary starts running. Other packers also mostly rely on these elementary steps.

Following those steps, the original binary contained in a packed program, no matter which packer was used, is first decrypted and then executed. In other words, an unpacking routine will write instructions that are executed later. Most generic unpacking methods including ours exploit that fact either directly or indirectly.

2.2 Packer Analysis Techniques

Single-step execution is a technique to monitor and disassemble each instruction of a packed program. With this technique, we can observe all instructions, trace the instruction pointer, and derive CPU registers’ values continually, while the program is unpacked on the memory. This technique is performed by customizing virtualization solutions such as Xen [16], KVM [17], and QEMU [18] for capturing the behavior at hypervisor level. It can be also performed by using dynamic binary instrumentation tools such as Intel Pin [19] and Valgrind [20].

Another technique consists in extracting written-and-executed fixed-size memory areas (i.e., memory pages). Some CPU architectures such as the Intel 64 and IA-32...
architectures [21] virtually divide a whole process memory into successive memory pages for page-level protection. For example, Intel IA-32 architecture adopts 4-Kbyte memory pages. On such architectures, we can mark memory pages as “read-only or writable” and “executable or not-executable” by changing read/write flags and executable-disable flags, respectively. Figure 2 describes the technique: we mark the whole process memory as read-only and executable after a packed program is loaded into the memory. Then, when a writing instruction is about to write some data to a read-only page, it causes a writing page-fault exception. At that time, we mark that page as writable and not-executable. After that, when an instruction on a not-executable page is about to be executed, it causes an executing page-fault exception. Thus we can recognize that the instruction is written and executed, at which time the memory page is marked as read-only and executable again. We call the memory page of the executed instruction a written-and-executed memory page.

Single-step execution enables an analyst to observe the whole behavior of a malware sample at a very fine-grained level. However, it induces a significant system performance overhead, not counting the necessity for the analyst to process large amounts of data to achieve her task. On the contrary, the Written-and-eXecuted (W&Xed) page extraction technique induces almost no overhead. However, she should observe the behavior at a coarse-grained level as it is only possible to take actions (e.g., inspect the memory or obtain the values stored in CPU registers) when a page-fault exception occurs.

2.3 Related Work

Several methods exploiting single-step execution have been proposed: PolyUnpack [7] aims at extracting the original binary of a given program packed by any packer. If a set of monitored instructions is not contained in the initial state of the packed program, it takes the set as newly generated instructions. Renovo [8] monitors jump instructions (e.g., ‘jmp’ and ‘je’) to detect the OEP of a packed program. If the instruction pointer moves to a dynamically generated instruction through a jump instruction, it determines that the instruction address is the OEP. Dinaburg et al. [9] propose an analysis environment named Ether based on Xen to thwart anti-analysis techniques that can detect analysis environments. They implement Renovo’s generic unpacking algorithm on Ether. Benninger et al. [22] and Deng et al. [23] independently propose that type of analysis environments by customizing Xen and KVM, respectively. Kawakoya et al. [10] focus on memory access ‘write’, ‘read’, and ‘execute’ of packed programs to detect the OEP, and Jeong et al. [24] focus on entropy scores in each section of a packed program on the memory to do so. Kim et al. [11] focus on a write-execute transition to spot more likely OEP candidates. Their system stores every written instruction, and it searches for a sequence of written instructions that have been executed successively. If such sequence is found, their system takes the beginning address of that sequence as an OEP candidate. This method supplies an unsorted set of OEP candidates.

OmniUnpack [12] aims at identifying malware programs with an anti-virus software scanner even if they are packed, while reducing the total number of scanning. To do this task efficiently, OmniUnpack adopts the W&Xed page extraction technique, and it scans newly generated memory pages only when a dangerous system call is executed (e.g., registry/network/file-write operations, process creation, etc.). The system is implemented with OllyBonE [25], which is a tool that supplies the W&Xed page extraction technique. OmniUnpack does not detect the OEP, so that we choose Justin for comparison in this paper.

2.4 Reference Work: Justin

Justin [13] adopts the W&Xed page extraction technique. It takes the addresses of written-and-executed instructions as OEP candidates to scan the whole memory with an anti-virus scanner, changing the starting scan address to each candidate. The main challenge of Justin is to detect the end of unpacking for reducing the number of OEP candidates, which results in making the number of scanning smaller. To this end, Justin possesses three heuristics to independently detect the end of unpacking (i.e., there are three variants of Justin). One is to check if a written memory page is part of the unpacking routine at the time an executing page-fault exception occurs on that memory page. Its efficiency for detecting the end of unpacking is a bit worse than the other types [13]. Another type (Justin_sp) focuses on the stack pointer, and the last type (Justin_ca) focuses on a set of command-line arguments.

The Justin_sp system memorizes the initial value of the stack pointer after a packed program is loaded into the process memory. Then it checks if the current value of the stack pointer is equal to the initial one every time an executing page-fault exception occurs. If it is not, the system takes the address of the instruction causing the page-fault exception as an OEP candidate; otherwise, Justin_sp determines that the unpacking routine has been already done, and it stops itself after taking an OEP candidate. This heuristic is based on the fact that most packers’ unpacking routines unwind the stack before the instruction pointer transfers to unpacked original binaries. Due to this, the original binaries need not
to care about having been packed.

The Justin_ca system checks if a set of command-line arguments is put on the stack every time an executing page-fault exception occurs. Then it takes the same operations as Justin_sp, where the set of command-line arguments is, for example, the argv of “int main( int argc, char *argv )”. This heuristic is based on the fact that some compiler-generated routines contained in the original binary put the set of arguments on the stack after the instruction pointer have passed through the OEP. Thus, if it is on the stack, the unpacking can be considered to be done.

3. Proposed Method

3.1 Basic Ideas

Our method adopts the W&Xed page extraction technique to quickly detect the OEP. In addition, in case our method fails to capture the OEP, it provides a sorted set of OEP candidates so that analysts will observe much less candidates.

As we explained in Sect. 2.1, the original binary of a packed program is written by the unpacking routine and executed later. In other words, the unpacking routine must consist of some memory pages that write instructions to be executed. Based on this observation, we store memory pages like page P in Fig. 2 as part of the unpacking routine, which generates an instruction to be executed. We also store memory pages that write some data to memory pages considered as part of the unpacking routine. This is because such memory pages can share the data with that unpacking routine and these memory pages can consist in it. We then select the executed instruction following the last instruction of the unpacking routine as the most likely OEP. After that, we provide all of the written-and-executed instructions as OEP candidates after sorting them.

3.2 Method

We define two types of memory pages that cause page-fault exceptions as follows: “generating page” is a memory page that writes some data into a memory page that is executed later, and “sharing page” is a memory page that writes some data to a generating page or a sharing page. In case 1 of Fig. 3, page E is a generating page and page F is a sharing page. We consider memory pages that are categorized into those two types as part of the unpacking routine, based on our basic ideas.

In order to classify memory pages, we observe the writing instruction when a writing page-fault exception occurs, at which time we store a pair of its src (source) address and dest (destination) address on array W. We also observe the written-and-executed instruction when an executing page-fault exception occurs, at which time we store its address on array X. We keep doing those operations until a packed program stops running or a predefined timeout is reached. Then we take each pair from W in chronological order, and check if the dest page is executed later or not by comparing it with X. If it is, we determine that the src page is a generating page, and we label the src address and its page as part of the unpacking routine (label U). After that, we take each pair from W in chronological order again, and check if the dest page has been labeled with U. If it is, we determine that the src page is a sharing page, and we label the src address and its page with U. We also label every address in X with U if it is on a memory pages of U. Finally, we list all addresses in X and src addresses in W in chronological order, and we pick up the addresses in X most closely following the last address labeled with U. This picked up address is considered as the most likely candidate.

For sorting candidates, we express all addresses in X in chronological order as $X = (x_0, x_1, \cdots, x_i, \cdots, x_p, \cdots, x_{l-1})$, where $x_p$ denotes the most likely OEP, $p$ its index, $x_{l-1}$ the last executed candidate, $l$ the number of the executed instructions, $i = 0, 1, 2, \cdots, l − 1$. We sort X based on $x_p$ so that candidates closer to $x_p$ are flagged as more likely, which is equivalent to using the following equation to calculate sorted indexes:

$$j = \begin{cases} 0 & (i = p) \\ 2 |p − i| − (1 + \text{sign}(p − i))/2 & (\text{otherwise}) \end{cases}$$

where, $j$ denotes a sorted index of $x_i$, $|\cdot|$ denotes absolute value, and $\text{sign}(\cdot)$ outputs $-1$ if the input is a negative integer; otherwise, outputs $1$. That is, our method sorts X as $(x_p, x_{p+1}, x_{p+2}, \cdots)$.

In summary, to get the sorted set of OEP candidates, we first store the pairs of src and dest address of writing instructions, and we also store the addresses of executed instructions. Then we detect the OEP as the most likely candidate among them. Finally, we sort the candidates based on the most likely candidate accordingly.

Case studies: in case 1 of Fig. 3, the address of the instruction at (5) is the most likely OEP, which most closely follows the last instruction labeled with U (i.e., the instruction at (4)). In case 2, page K is a generating page, and the address of the instruction at (3) is the most likely OEP.
4. Evaluation

To evaluate our method, Justin_sp and Justin_ca, we independently apply those methods to a dataset of 753 samples packed by 25 packers, and we try to provide OEP candidate sets in this section. Then we evaluate the methods with a recall measure, in terms of reducing candidates and uniquely locating the OEP. After that, we propose a method combining our method with Justin_sp. Finally, we evaluate effectiveness of our method, Justin_sp, and the combined method over the whole dataset with an F-measure graph and a recall-precision graph.

4.1 Dataset and Environment

We used 35 malware samples chosen as follows. We collected malware programs among the past few years, and they were categorized into 135 malware categories such as 'Backdoor.IRC.Bot' and 'Unknown' by a Symantec antivirus scanner. We took the top 35 categories by number of samples except for the unknown category. We then checked if the samples in the categories were packed or not in order to take not-packed samples, whose entry points can be obtained exactly. To this end, we collected compiler signatures for PEiD, a signature-based packer-and-compiler identification tool. Besides, we used Lyda et al.'s method [26], which can identify not-packed programs with an entropy analysis. After that, we randomly took one sample per category that was not packed by any packer. The SHA-256 [27] hash values of the 35 samples are different from each other, and Table 3 shows the malware names.

We used 25 packers to pack each of the 35 malware samples, and we discarded packed samples that did not run due to failure of packing (e.g., the trial version of exe32pack cannot be used to pack binaries that are larger than 80 KB). We eventually obtained 753 packed variants from those 35 initial samples. Please refer to Table 1 for a breakdown of the number of valid samples that were obtained by each packer (cf., 2nd column).

As an environment for the experiments, we prepared Windows XP as a guest OS in VirtualBox [28], and we implemented our method, Justin_sp, and Justin_ca through a kernel driver. We then executed each method independently to list OEP candidates for each sample. It is to be noted that while we relied on Windows XP in our experiments, the same principles should be applicable to more recent versions of Microsoft Windows.

4.2 Overview of Results

Using the W&Xed page extraction technique, we picked up a set of OEP candidates for every packed sample. The average number of candidates for each packer's samples is shown in Column 4 of Table 1, and the standard deviation is shown in Column 5. We verified that all the sets of candidates contained the correct OEPs.

Our method sorted those candidate sets. To evaluate

| Packer          | # of candidates | Recall (%) |
|-----------------|-----------------|------------|
| ASPack 2.33     | 9.23            | 94%        |
| ASProtect 1.70  | 67.20           | 97%        |
| exe32pack 1.42 trial | 7.00       | 97%        |
| Exe Stealth 2.73 trial | 9.43      | 97%        |
| Ezip 1.0        | 10.00           | 97%        |
| FSF 2.0         | 8.76            | 97%        |
| Mew11SE 1.2    | 11.42           | 97%        |
| MoleBoxPro 2.6.4 trial | 23.29     | 97%        |
| mpress 2.19    | 9.87            | 97%        |
| nPack 1.1.300  | 9.39            | 97%        |
| NsPack 3.7 trial | 10.00          | 97%        |
| Packman 1.0    | 9.29            | 97%        |
| PECCompact 2.79 trial | 10.94      | 97%        |
| PESpin 1.33    | 13.12           | 97%        |
| Petite 1.4     | 11.56           | 97%        |
| PKLITE32.1.1   | 9.57            | 97%        |
| RL.Pack 1.20   | 10.00           | 97%        |
| SimplePack 1.0 | 10.00           | 97%        |
| tElock 0.99    | 8.42            | 95%        |
| Themida 2.2.7.0 | 300.47          | 97%        |
| Upack 0.399    | 11.73           | 97%        |
| UPX 3.08       | 10.09           | 97%        |
| WinUpack 0.31  | 10.21           | 97%        |
| WWPack32 1.20 trial | 9.18      | 97%        |
| yoda’s protector 1.02 | 14.71    | 97%        |

Table 1 The results of our method. #: the number of variants created by the packer, # of candidates: the number of the candidates that the extraction technique picked up, 'AVG': the average of # of candidates, 'SD': the standard deviation of # of candidates, recall: the percentage of success results and a result is a success if the OEP is contained within the top n candidates, '-' : 100%.

Total 753 Average 81 85 87 92 96 97 99
Table 2  The results of Justin\_sp. # of candidates: the number of candidates that Justin\_sp derived. ’AVG’: the average of # of candidates, ’SD’: the standard deviation of # of candidates, recall: the percentage of candidate sets that contain their OEPs.

| No. | Packer               | #     | # of candidates | AVG | SD (%) | Recall |
|-----|----------------------|-------|-----------------|-----|--------|--------|
| 1   | ASProtect 2.33       | 35    | 1.00            | 0.00| 100    |        |
| 2   | ASProtect 1.70       | 35    | 65.29           | 12.86| 100    |        |
| 3   | exe32pack 1.42 trial| 10    | 2.90            | 0.70| 100    |        |
| 4   | Exe Stealth 2.73 trial| 35    | 1.00            | 0.00| 100    |        |
| 5   | Ezip 1.0             | 35    | 1.00            | 0.00| 100    |        |
| 6   | FSG 2.0              | 34    | 1.00            | 0.00| 100    |        |
| 7   | MewISLE 1.2          | 33    | 5.03            | 7.40| 100    |        |
| 8   | MoleBoxPro 2.64 trial| 35    | 14.66           | 0.47| 100    |        |
| 9   | npress 2.19          | 31    | 2.00            | 0.00| 100    |        |
| 10  | nPack 1.1.300        | 33    | 1.00            | 0.00| 100    |        |
| 11  | NoPack 3.7 trial     | 34    | 1.06            | 0.24| 100    |        |
| 12  | Packman 1.0          | 35    | 1.03            | 17.00| 100    |        |
| 13  | PECompact 2.79 trial| 34    | 2.03            | 0.17| 100    |        |
| 14  | PESpin 1.33          | 34    | 5.00            | 0.00| 100    |        |
| 15  | Petite 1.4           | 18    | 2.56            | 0.50| 100    |        |
| 16  | PKLITE32 1.1         | 14    | 1.00            | 0.00| 14     |        |
| 17  | RLPack 1.20          | 34    | 6.09            | 13.35| 100    |        |
| 18  | SimplePack 1.0       | 33    | 1.00            | 0.00| 100    |        |
| 19  | tElock 0.99         | 19    | 2.00            | 0.00| 100    |        |
| 20  | Themida 2.2.7.0      | 32    | 1.00            | 0.00| 100    |        |
| 21  | Upack 0.399         | 26    | 1.88            | 0.42| 00     |        |
| 22  | UPX 3.08            | 34    | 1.00            | 0.00| 100    |        |
| 23  | WinUpack 0.31       | 33    | 1.15            | 0.36| 100    |        |
| 24  | WWPack32 1.20 trial | 22    | 1.00            | 0.00| 100    |        |
| 25  | yoda’s protector 1.02| 35    | 14.71           | 5.58| 100    |        |

Average 6.06 14.36 94

Table 3  The percentage of variants against which the Justin\_ca system recognized a set of command-line arguments put on the stack, where # denotes the number of ones that were packed by the 25 packers, except for variants that suffer from packing failure.

| Malware                  | # | %     |
|--------------------------|---|-------|
| Backdoor:IRC.Bot         | 23| 9     |
| Backdoor:Notol           | 18| 100   |
| Backdoor:Sdbot           | 23| 4     |
| Backdoor:Trojan          | 17| 6     |
| Downloader               | 21| 100   |
| Hacktool                 | 23| 26    |
| Infostealer              | 21| 90    |
| Infostealer.Gampass      | 22| 100   |
| Trojan.Botime            | 19| 5     |
| Trojan.Dropper           | 23| 17    |
| Trojan.Farli             | 22| 5     |
| Trojan.Gen               | 22| 95    |
| Trojan.Gen.2             | 23| 91    |
| Trojan.Panddos           | 24| 100   |
| Trojan.Stabuniq          | 22| 95    |
| Trojan.Usuage!gen3       | 21| 95    |
| Trojan.Horse             | 23| 9     |
| W32.Bobax’dr             | 21| 29    |

Average 54

sample packed by exe32pack, and all the sets contain their correct OEPs. As a result, the recall is 100%. In Table 2, we can see that Justin\_sp provided 6.06 candidates on average with a recall of 94%. In addition, Justin\_sp derived only one candidate for every sample of ASProtect (#1), EzeStealth (#4), Ezip (#5), FSG (#6), nPack (#10), SimplePack (#18), UPX (#22), and WWPack32 (#24) with a recall of 100%.

Our implementation of Justin\_ca failed to capture likely candidates for many malware variants. This is because their original malware programs removed a set of command-line arguments from the stack after it was put on the stack. Suffering from this, Justin\_ca did not even recognize when a set of arguments was put on the stack unless an executing page-fault exception occurred between the arguments-putting operation and the stack-removing operation. Table 3 shows the percentage of variants against which Justin\_ca recognized the set of arguments put on the stack. It is just 54% of all the variants. In the following sections, we omit discussions of Justin\_ca.

4.3 Comparison between Heuristics

As shown in Table 2, the Justin\_sp system provided 6.06 candidates on average with a recall of 94%, whereas our method gathered the correct OEP within the top six candidates (i.e., n = 6) with a recall of 92% on average, as shown in Table 1. Justin\_sp’s recall is better than ours by 2%. However, our method uniquely located correct OEPs of a large part of the success results for n = 6. In other words, a recall of 81% for n = 1 means that our method uniquely located the OEP against 610 packed samples, which occupies 88% of the success results for n = 6. Justin\_sp does not consider sorting, but it uniquely located correct OEPs just for the eight packers described in the previous section.

Table 2 shows that Justin\_sp did not work well for samples packed by ASProtect, PKLITE32, Themida, and

---

1In general, recall is defined as the percentage of items that are relevant and in fact retrieved [29]. We consider relevant items as the OEPs of samples in our experiments.
yoda’s protector. The results of ASProtect and yoda’s protector can be explained by the fact that those packers do not entirely clean up the stack before the instruction pointer jumps to the OEP. Then the samples packed by Themida or PKLITE32 had set up the stack pointer value as the initial one when the first executing page-fault exception occurred. Suffering from this, Justin_sp took only one candidate for every sample, but all the candidates were wrong for the Themida, and just 14% of the candidates were correct for PKLITE32. Those results indicate that Justin_sp can discard correct OEPs.

Our method could not get good results for ASProtect. This can be explained by the fact that ASProtect uses complicated routines that keep unpacking part of the original binary and executing it in turn little by little. In other words, although the instruction pointer has passed through the OEP, some writing instructions write other instructions to be executed for unpacking the original binary. Such complicated routines decreased the recall of our method. However, Table 1 shows that our method was able to reduce from 67.20 candidates on average to 32 candidates (i.e., n = 32) with a recall of 86%. In addition, our method could achieve a recall of 37% when n equals one in spite of such complicated routines. Those results were caused by the fact that just after original binaries had been unpacked, they waited for some conditions to be satisfied such as receiving commands from a command-and-control (C&C) server. Thus the complicated routines were not processed by chance. As other remarkable results, our method could also reduce from 300.47 candidates to 16 candidates (i.e., n = 16) for Themida. Then our method uniquely located OEPs (i.e., n = 1) with a recall of 97% for yoda’s protector, and it achieved a recall of 100% for PKLITE32 when n equals two.

Therefore, from the perspective of the comparisons in this section, we can tell that our method can be more effective than Justin_sp. By using our method, analysts can observe much less OEP candidates when analyzing malware, which brings about more effective analysis.

4.4 Discussion for Obtaining Better Results

We examine why our method could not achieve a recall of 100% for the packers in Table 1 when n equals one, excluding ASProtect, MoleBoxPro, Petite, PKLITE32, Themida, and Upack. The main reason is that our method did not work efficiently for several variants of ‘Trojan.Gen.2’ and those of ‘W32.Virut.R’. The two types of variants wrote some data to the memory pages that contained their OEPs even though the instruction pointer had passed through the OEPs. Our method considered those writing instructions as part of their unpacking routines.

As shown in Table 2, the Justin_sp system took more than one candidate for several packers, except for the packers discussed in the previous section. Those results were caused by several executing page-fault exceptions that happened before their unpacking routines cleaned up the stack. Justin_sp is directly affected by such page-fault exceptions. However, Justin_sp cannot be affected by variants that can degrade our results. That is, the clean-up operation is generally performed before some complicated routines of packed malware are executed.

The results that we obtained via our method and Justin_sp differed significantly for certain types of packers or malware. We sensed that they could supplement each other if they were combined properly.

4.5 Combined Method

We propose a method combining our method with Justin_sp. It focuses on the common candidates between Justin_sp’s unsorted set and our sorted set, taking them as more likely OEPs as follows. By independently applying Justin_sp and our method to a target program, we obtain two OEP candidate sets. Then we take k candidates from the head of our sorted set, and store them on array α, where k is a given parameter and k = 1, 2, · · · . At which time, we give a small value to k so that we can take candidates finely closer to the head. This means that we will not use less likely candidates. After that, we extract candidates contained in Justin_sp’s set from α, keeping α’s order. We then add the remaining of α to the tail of those extracted candidates, which we store on array β. We finally add the rest of our sorted set to the tail of β, which is the final result of the combined method.

We calculate the recall of the combined method, changing parameter k until 12, where the definition of recall is the same as defined for our method in Sect. 4.2. Then we show the average recall for all the 753 packed samples in Fig. 5, where n = 1, 2, · · · , 7. The recall values increase until k reaches six if n is smaller than six; otherwise, the recall is the same as our method regardless of k. Considering the recall for n = 1 and k = 6, we confirm that the combined method improved the recall of our method by 5%, which is one of the best results for n = 1.

We focus on the results for n = 1 and k = 6, and list the candidates by W&Xed page extraction.

| Candidates by W&Xed page extraction |
|------------------------------------|
| (a, b, c, d, e, f) |
| sorted by ours (k=4) derived by Justin_sp |
| (d, c, e, b, f, a) (a, b, c) |
| α: (d, c, e, b) |
| β: (c, b, d, e) |
| Final result: (c, b, d, e, f, a) |

Fig. 4 An example for obtaining a result sorted by the combined method (#candidates = 6, k = 4).

†When k equals one, the recall values of the combined method are the same as those of our method because it takes only the first candidate from our sorted set.
recall for each packer in Table 4. The combined method improved or retained the recall of ours for every packer. In particular, it achieved a recall of 100% for 16 packers, for which our method could not achieve 100%. Compared to Justin’s results in Table 2, the combined method got better results for exec32pack, Mew11SE, NsPack, Packman, PECOMPACT, RLPack, tElock, and WinUpack. That is, the combined method uniquely located the correct OEPs with a recall of 100% for those eight packers, whereas Justin’s offered more than one candidate. Then the results for mpress, PESpin, and Upack can be better than those of Justin’s because the combined method achieved more than 95%. In addition, the combined method achieved a recall of 97% for yoda’s protector, but Justin’s could not capture likely candidates at all. Thus we can tell that the combined method can be more effective than either our method or Justin’s.

4.6 Effectiveness over the Whole Dataset

To evaluate effectiveness of the three methods over the whole dataset, we generate two graphs in this section: an F-measure graph and a recall-precision graph. To this end, we define precision as the percentage of checked candidates that are in fact OEPs. For example, if there is a candidate set and an analyst retrieves five candidates from that set (i.e., \( n = 5 \)), a precision of 20% \((1/5)\) means that she checks five candidates and the OEP is contained in them; if the precision is 25% \((1/4)\), \( n = 4 \). As this example indicates, the precision increases as \( n \) decreases as long as the OEP is contained in the top \( n \) candidates. This means that the precision measures how well a method is doing at reducing the size of false-positive (not-OEP) candidates. Unlike the precision, the recall increases (or keeps its value) as \( n \) also increases. That is, there is a trade-off between the recall and the precision, which is often used for evaluating effectiveness in some search applications.

The F-measure is an effectiveness measure based on recall and precision [30]. This measure can summarize effectiveness in a single-number. It is defined as the harmonic mean of recall \( R \) and precision \( P \) as \( F = 2RP/(R+P) \). The F-measure graph that we generate in this section is plotted by calculating \( F \) for each \( n \). In addition to this effectiveness summary, the recall-precision graph is generated by plotting recall-precision values to show their relations as data point \((R, P)\).

Our method and the combined method sorted the 753 candidate sets obtained in Sect. 4.2. Then we calculated the recall-precision value \((R_n, P_n)\) for each set, incrementing \( n \) by one, where \( m \) denotes an index of a set (i.e., \( m = 1, 2, \ldots, 753 \)). After that, we calculated the average recall-precision value \((R_m, P_m)\) for all the sets as \(((R_1, P_1) + \cdots + (R_m, P_m))/m)\). If \( n \) is larger than the number of candidates of the \( m \)-th set, we used \((R_m, P_m)\) instead, where \( s \) denotes the number of candidates of the \( m \)-th set. In this case, \( R_s \) is 1.0 and the precision is \( 1/s \) since all candidates of the \( m \)-th set are checked.

Because the Justin system outputs the unsorted sets shown in Table 2, we cannot directly calculate \((R_n, P_n)\) from Justin’s results. To do this, we define a model of an analyst who tries to obtain the OEP from an unsorted set. First, Justin’s outputs an unsorted set that has \( y \) candidates, taking as input a malware sample. We call this set ‘the first set Y’. This corresponds to the candidate sets shown in Table 2. In addition to Y, Justin’s also outputs all candidates obtained with the page extraction technique, excluding the candidates contained in the first set Y. We call those candidates ‘the second set Z’, and we define \( z \) as the number of candidates contained in Z. Then an analyst retrieves and ex-

![Fig. 5 The recall of the combined method (\( n = 1, \ldots, 7 \) and \( k = 1, \ldots, 12 \)).](image)}
Fig. 6 An F-measure graph for our method, Justin_sp, and the combined method.

includes a candidate from the first set Y at random. She keeps this until all candidates are excluded from the first set Y. After that, she retrieves and excludes a candidate from the second set Z at random. She keeps this until all candidates are excluded from the second set Z.

To check a probability w that an analyst will obtain the OEP, we actually worked following this model against the 753 malware samples several times. We then confirmed that w approximately equals n/y if the OEP is contained in the first set Y, where n denotes the number of checked candidates. If the OEP is contained in the second set Z and n ≤ y, w equals zero. If n > y, w approximately equals (n − y)/y. Those results can be explained by probability theory. After that, we also confirmed that the average recall R_n approximately equals w. This is because the recall measures whether an analyst obtains the OEP when she takes n candidates. This is obviously determined by w. In addition, the average precision P_n approximately equals w/n because the precision is zero if the OEP is not contained in the n candidates of a set; otherwise, the precision equals 1/n. That is, P_n approximately equals its expected value (i.e., 0 × (1 − w) + 1/n × w). Thus we calculated w and w/n as (R_n, P_n) of Justin_sp for the 753 candidate sets.

We calculate F-measure for each n as F_n = 2R_nP_n/(R_n + P_n), and we plot these data points on the graph in Fig. 6 for the three methods. Figure 6 confirms that our method has better effectiveness over the whole dataset than Justin_sp. In particular, when n equals one or two, the gap of F between those two methods is relatively larger. This means that our method is very effective when an analyst retrieves one or two candidates from a set. In addition, the combined method improves Justin_sp overall, and it improves our method until n equals six.

We then obtain the precision P at any recall level R for the recall-precision graph, we use the following interpolation technique:

\[
P(R) = \max \{ P' \mid R' \geq R \land (R', P') \in S \}
\]

where, \( \land \) and \( S \) denote logical conjunction and the set of observed \((R,P)\) points, respectively. This interpolation defines the precision at any recall level as the maximum precision observed in any recall-precision point at a higher recall level.[30] We plot those obtained data points on the graph in Fig. 7. This graph also confirms that our method is more effective than Justin_sp overall. Justin_sp’s results are degraded especially by ASProtect, PKLITE32, Themida, and yoda’s protector. For example, it results from the fact that an analyst retrieves a candidate from the second set Z that has around 300 candidates for a sample packed by Themida. Then we can see that the combined method improves our method and Justin_sp. In particular, the combined method achieved a recall of 85.7% with a precision of 85.7%, whereas the precision of our method is 29.9% when the recall is 85.7%. The combined method succeeded in reducing the size of candidates significantly.

4.7 Performance Overhead

The system of our method mainly consists of two modules: the page-extraction module and the candidate-sorting module. The first module stores the instruction addresses with the W&Xed page extraction technique. The other module provides candidate sets with the OEP detection algorithm and the candidate-sorting one. We check how much performance overhead those modules will cause.
We executed several malware variants with our system five times on a 3.4GHz Intel-Core-i7 machine. At that time, when executing a variant, we measured the elapsed time taken from the start of the variant until the end of its first process. In addition, we also did it without our system five times. The chosen variants were 24 packed samples of Trojan.Panddos and the Trojan.Usuge!gen3 sample packed by PKLITE32. The reason why we chose Trojan.Panddos is that the number of packed variants of Trojan.Panddos is largest in those of the other original samples shown in Table 3. We used the Trojan.Usuge!gen3 sample packed by PKLITE32 instead because only PKLITE32 had failed to pack a sample of Trojan.Panddos. In this evaluation, we define performance overhead as a time margin between the elapsed time of a variant with and without our system.

Table 5 shows the average elapsed time of five measurements and the average of its corresponding time margin (i.e., performance overhead) for 25 packed samples. ‘Original time’ in the table means the elapsed time of a variant measured without our system. ‘Time margin’ means a margin between the original time and the elapsed time measured with our system. Overall, our system did not introduce a large amount of overhead. In particular, the overhead was very small such that the elapsed time measured with our system became smaller than the original time against several packers (e.g., exe32pack and FSG).

The largest amount of overhead occurred under

![Image]

| Packers used for samples | Size (KB) | Original time (msec) | Time margin (msec) | (%)  |
|--------------------------|----------|----------------------|-------------------|-----|
| ASPack                   | 29       | 103                  | 5                 | 4.9 |
| ASProtect                | 176      | 153                  | 5                 | 3.3 |
| exe32pack                | 30       | 103                  | –3                | –2.9|
| Exe Stealth              | 69       | 99                   | 13                | 13.1|
| Ezip                     | 69       | 101                  | 4                 | 4.0 |
| FSG                      | 24       | 100                  | –1                | –1.0|
| Mew11SE                  | 24       | 102                  | 5                 | 4.9 |
| MoleBoxPro               | 94       | 119                  | 12                | 10.1|
| mpress                   | 26       | 109                  | –1                | –0.9|
| nPack                    | 29       | 98                   | 9                 | 9.2 |
| NoPack                   | 25       | 100                  | 12                | 12.0|
| Packman                  | 24       | 94                   | 9                 | 9.6 |
| PECCompact               | 26       | 107                  | –2                | –1.9|
| PESpin                   | 47       | 130                  | 3                 | 2.3 |
| Petite                   | 31       | 99                   | 4                 | 4.0 |
| PKLITE32                 | 111      | 116                  | –6                | –16.7|
| RLPack                   | 24       | 97                   | 4                 | 4.1 |
| SimplePack               | 25       | 98                   | 0                 | 0.0 |
| tElock                   | 38       | 107                  | 5                 | 4.7 |
| Themida                  | 1184     | 1220                 | 41                | 3.4 |
| Upack                    | 22       | 107                  | 5                 | 4.7 |
| UPX                      | 26       | 93                   | 8                 | 8.6 |
| WinUpack                 | 22       | 103                  | 12                | 11.7|
| WWPack32                 | 36       | 103                  | 5                 | 4.9 |
| yoda’s protector          | 44       | 6399                 | 33                | 0.5 |

*Themida*, and it was 41 milliseconds. *Themida* caused page-fault exceptions more frequently than the other packers. This led to increasing the number of addresses stored by the page-extraction module. Taking as input those addresses, the sorting-candidate module spent 14 milliseconds on detecting the most likely OEP. The rest of overhead (i.e., 27 milliseconds) was caused by the page-extraction module. Under the other packers, the sorting-candidate module performed its task within less than 2 milliseconds per sample.

We measured how much time our system spent on providing the candidate sets of the 753 samples. The total time consumption was 1061 seconds, and its average per sample was 1.41 seconds. The time consumption of our system depends on the original time of malware samples. However, in terms of low performance overhead, we can tell that our system can be efficient. Finally, we performed the same evaluation for the system of the combined method. We also confirmed that the overhead caused by that system was very small as well as our system.

5. Conclusion

On the one hand, both the Justin_sp system and our method employ the W&Xed page extraction technique in order to achieve their tasks quickly, compared to existing methods relying on single-step execution. On the other hand, there exist two major differences between the Justin_sp system and our method: one is heuristics for detecting the end of unpacking and the other difference is to sort candidates, in terms of reducing the number of OEP candidates. That is, our method focuses on relations between writing instructions and written areas, whereas Justin_sp focuses on the stack-clean-up operation conducted by the unpacking routine. Then our method can provide a sorted set of candidates ranked by the most likely one. We can tell from the comparisons conducted in Sects. 4.3 and 4.6 that our method can be more effective than Justin_sp.

We also proposed a method combining our method with Justin_sp, and we confirmed that it improves the two. This result shows that our method can support other generic unpacking methods. In particular, the combined method uniquely located the correct OEP for 86% of all the packed malware samples. By using the combined method, analysts can be expected to get the correct OEP more quickly. In our future work, we have to study how to defeat some packers that possess complicated packing algorithms such as ASProtect and Themida.

Acknowledgments

The authors would like to thank Erwan Le Malécot, a colleague of ours, for his valuable comments and suggestions toward a clearer formulation of the proposed scheme(s), and for his further editorial assistance.

References

[1] M. Sikorski and A. Honig, Practical Malware Analysis: The Hands-
On Guide To Dissecting Malicious Software, No Starch Press, 2012.

[2] K.A. Roundy and B.P. Miller, “Binary-code obfuscations in prevalent packer tools,” ACM Comput. Surv., vol.46, no.1, pp.4:1–4:32, July 2013.

[3] M.F. Oberhumer, L. Molnar, and J.F. Reiser, “UPX: Ultimate packer for executables,” http://upx.sourceforge.net/, 2014.

[4] StarForce Technologies Ltd., “ASPack Software,” http://www.aspack.com/, 2014.

[5] Bitsum LLC, “Peachparent,” https://bitsum.com/peachparent/, 2014.

[6] Orenäs Technologies, “Themida,” http://www.orenäs.com/themida.php, 2014.

[7] F. Guo, P. Ferrie, and T.C. Chiueh, “A study of the packer problem: Automating the hidden-code extraction of unpack-executing malware,” Proc. 22nd Annual Computer Security Applications Conference (ACSAC ‘06), pp.289–300, 2006.

[8] M.G. Kang, P. Poosankam, and H. Yin, “Reno: A hidden code extractor for packed executables,” Proc. 5th ACM Workshop on Recurring Malcode (WORM’07), pp.46–53, New York, NY, USA, ACM, 2007.

[9] 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’08), pp.51–62, New York, NY, USA, ACM, 2008.

[10] Y. Kawakoya, M. Iwamura, and M. Itoh, “Memory behavior-based automatic malware unpacking in stealth debugging environment,” Proc. 5th International Conference on Malicious and Unwanted Software (MALWARE ’10), pp.39–46, 2010.

[11] H.C. Kim, T. Orii, K. Yoshioka, D. Inoue, J. Song, M. Eto, J. Shikata, T. Matsumoto, and K. Nakao, “An empirical evaluation of an unpacking method implemented with dynamic binary instrumentation,” IEICE Trans. Inf. & Syst., vol.E94-D, no.9, pp.1778–1791, Sept. 2011.

[12] L. Martignoni, M. Cristodorescu, and S. Jha, “OmnimPack: Fast, generic, and safe unpacking of malware,” Proc. 23rd Annual Computer Security Applications Conference (ACSAC ’07), pp.431–441, 2007.

[13] F. Guo, P. Ferrie, and T.C. Chiueh, “A study of the packer problem and its solutions,” Proc. 11th International Symposium on Recent Advances in Intrusion Detection (RAID ’08), pp.98–115, Springer-Verlag, Berlin, Heidelberg, 2008.

[14] M. Ligh, S. Adair, B. Hartstein, and M. Richard, Malware Analyst’s Cookbook and DVD: Tools and Techniques for Fighting Malicious Code, Wiley, 2010.

[15] E. Gele, T. Scholte, E. Kirda, and C. Kruegel, “A survey on automated dynamic malware-analysis techniques and tools,” ACM Comput. Surv., vol.44, no.2, pp.6:1–6:42, March 2008.

[16] Xen Project, “The Xen Project,” http://www.xenproject.org/, 2013.

[17] Red Hat Open Source Community, “KVM: Kernel-based virtual machine,” http://www.linux-kvm.org/.

[18] F. Bellard, “QEMU,” www.qemu.org/

[19] Intel Corporation, “Pin — A dynamic binary instrumentation tool,” available at http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

[20] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” SIGPLAN Not., vol.42, no.6, pp.89–100, June 2007.

[21] Intel Corporation, “Intel 64 and ia-32 architectures software developer’s manual,” available at http://www.intel.com/content/dam/www/public/us/en/documents manuals /64-ia-32-architectures-software-developer-manual-325462.pdf, 2014.

[22] C. Benninger, S. Neville, Y. Yazir, C. Matthews, and Y. Coady, “Maitland: Lighter-weight VM introspection to support cybersecurity in the cloud,” Proc. 5th IEEE International Conference on Cloud Computing (CLOUD ’12), pp.471–478, 2012.

[23] Z. Deng, X. Zhang, and D. Xu, “SPIDER: Stealthy binary program instrumentation and debugging via hardware virtualization,” Proc. 29th Annual Computer Security Applications Conference (ACSAC ’13), pp.289–298, 2013.

[24] G. Jeong, E. Choo, J. Lee, M. Bat-Erdene, and H. Lee, “Generic unpacking using entropy analysis,” Proc. 5th International Conference on Malicious and Unwanted Software, MALWARE ’10, pp.98–105, 2010.

[25] J. Stewart, “Ollybone v0.1, break-on-execute for ollydbg (html document),” http://www.joestewart.org/ollybone/tutorial.html, 2006.

[26] R. Lyda and J. Hamrock, “Using entropy analysis to find encrypted and packed malware,” IEEE Security and Privacy, vol.5, no.2, pp.40–45, March 2007.

[27] P.O.J. Bryson, “Secure hash standard (shs) (federal information processing standards publication 180-4),” available at http://csrc.nist.gov/publications/fips/fips180-4/fips180-4.pdf, 2012.

[28] Oracle, “Oracle VM VirtualBox,” https://www.virtualbox.org/, 2014.

[29] J. Han, M. Kamber, and J. Pei, Data Mining: Concepts and Techniques, Third ed., Morgan Kaufmann, 2011.

[30] B. Croft, D. Metzler, and T. Strohman, Search Engines: Information Retrieval in Practice, Addison-Wesley, 2009.

Ryoichi Isawa received his B.E. and M.E. degrees from the University of Tokushima, Japan, in 2004 and 2006, respectively. He received his Ph.D. degree from Kobe University, Japan, in 2012. He is currently a researcher at the National Institute of Information and Communications Technology (NICT), Japan. His current research interests include malware analysis, network security, and information security. He is a member of the IEEE and the Information Processing Society of Japan.

Daisuke Inoue received his B.E. and M.E. degrees in electrical and computer engineering and Ph.D. degree in engineering from Yokohama National University in 1998, 2000, and 2003, respectively. He joined the Communications Research Laboratory (CRL), Japan, in 2003. The CRL was relaunched as the NICT in 2004, where he is the director of Cybersecurity Laboratory in NSRC.

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.