Original Entry Point detection based on graph similarity

Thanh-Hung Pham[1] and Mizuhito Ogawa[1]

Japan Advanced Institute of Science and Technology, Japan
hung.pthanh@gmail.com and mizuhito@jaist.ac.jp

Abstract. This paper proposes a method for packer identification and OEP (Original Entry Point) detection based on the graph similarity on control flow graphs of packed codes. Packed code consists of an unpacking stub and a packed payload, which is recovered to the original after the unpacking stub executes. In this paper, the CFGs of packed code are generated by a DSE (Dynamic Symbolic Execution) tool BE-PUM on x86-32/Windows. We define the *template* of the unpacking stub as the pair of the average of Weisfeiler-Lehman histogram vectors and the end sequence. Next, each template is computed packer-wise (i.e., processing packed codes by the same packer) for the ease of covering a new packer. We use the total of 71 samples packed by 12 packers. For unknown packed code, we will find the templates in its CFG generated by BE-PUM.
Among them, the CFG fragment with the highest cosine similarity is regarded as the unpacking stub, which also detects the used packer and the OEP as the jump destination from the exit.
Our first experiment is performed on 700 non-malware samples (of which the original payload is also known) packed by 12 packers above. The used packer is correctly identified for 689 and the OEP is correctly detected for 688. Further, we apply the method to 1239 malware samples. Among them, 1089 samples are detected packed by *unknown packer* and among them 150 samples are detected as packed by the 11 packers (except for TELOCK) and their OEPs are detected. We conclude that our method is highly effective as long as we have access to an executable of a target packer to compute its templates.

Keywords: Original Entry point detection · Packer Identification· Graph similarity.

1 Introduction

Malware threat increases every year. Not only new techniques are introduced, but also a systematic development becomes popular, such as the use of a packer. It is said that more than 80% of recent malware is obfuscated by packers to bypass anti-virus software. For x86, more than 50 popular packers are available on the net, and they often encrypt the payload which is decrypted at runtime by unpacking stubs (Fig 1). Hence, the detection of the used packer and the OEP

(Original Entry Point) is important to understand the hidden actions of the payload of malware. Control obfuscations, such as indirect jumps, code flattening, opaque predicates (mixing dead code), and self-modification, are mostly in the unpacking stub. It is believed that DSE (Dynamic Symbolic Execution) [1, 2] is the most powerful de-obfuscation, which exhaustively traces feasible control paths only. We adopt a DSE tool BE-PUM [3] on x86-32/Windows to obtain precise control flow graphs of packed code.

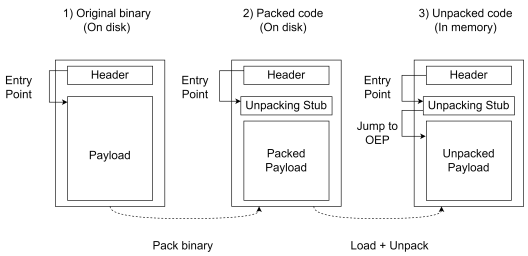

Fig. 1: Packing and Unpacking Process

This paper proposes a method for the packer identification and the OEP (Original Entry Point) detection based on the graph similarity of CFGs (Control Flow Graphs) of packed codes. Packed code consists of an unpacking stub and a packed payload, which is recovered to the original after the unpacking stub executes. We start with the hypothesis that the CFG of the unpacking stub characterizes a packer. Note that when the original payloads and used packers are known (e.g., pack the payload by ourselves) we can identify the unpacking stub. Fig 4 (Section 4) shows preliminary observation on UPX, FSG, and Mew, respectively. A packer may have different patterns of CFGs of the unpacking stubs (e.g., WINUPACK has 2), but the pattern of the unpacking stubs converges regardless of the payloads. Our tool for the graph similarity is a *Weisfeiler-Lehman histogram vector* (Section 3) of a CFG, which is computed iteratively by relabeling each node with the collection of neighbor's labels [4].

First, to confirm the hypothesis, we classify 771 samples packed by 12 packers (UPX v3.95, ASPACK v2.12, FSG v1.0, YODA v1.3, MEW SE v1.2, PACKMAN v1.0, PECOMPACT v2.xx, PETITE v2.1, WINUPACK v0.39 Final, JDPACK v1.01, MPRESS v2.xx, and TELOCK v0.98) by a clustering algorithm DBSCAN wrt the graph similarity, (which is the cosine similarity [5] on Weisfeiler-Lehman histogram vectors). We observe that when the allowance *eps* of DBSCAN is small enough (e.g., $eps = 0.02$), (1) each class does not cross different packers, and (2) the end sequence (the prefix of the exit of an unpacking stub) is the same in each class. Hence, we define the *template* of the class as the pair of the average of Weisfeiler-Lehman histogram vectors (of CFGs of the unpacking stubs) and the end sequence. The length of the vector rapidly increases with respect to the number k of the diameter, and we set $k = 2$.

Next, we prepare the templates of 12 packers, using 71 packed samples. For the ease of introducing a new packer, we compute them packer-wise (i.e., clustering code packed by the same packer). Finally, for an unknown code, BE-PUM incrementally generates the CFG. When the end sequence matches with the tail

of a CFG fragment, we check the similarity between its Weisfeiler-Lehman histogram vector and that in prepared templates. Among them, the CFG fragment with the highest cosine similarity is regarded as the unpacking stub, which also detects the used packer and the OEP as the jump destination from the exit.

We perform experiments on 700 non-malware samples (which are packed by 12 packers mentioned above). Our method correctly detects the packer for 689, and the OEP for 688. For the packer identification, `VirusTotal` (a database collected from various resources) identifies 699 beyond our result, but the OEP detection result is distinguished from others, e.g., `GUnpacker` and `QuickUnpack` find 525 and 283, respectively. Further, 1239 malware samples are examined. Among them, 1089 are detected packed by *unknown packer* and 150 are packed by the 11 packers (except for TELOCK). Our main contributions are,

1. Apart from dynamic analysis based on dirty page tracing, we observe the control flow generated by DSE tool BE-PUM [3] (when we have access to an executable of a target packer).
2. We combine statistical similarity (Weisfeiler-Lehman kernel) with the symbolic evidence (the end sequence of the unpacking stub) to characterize the unpacking stub. This pair is called the *template* of a packer.

The paper is constructed as follows. Section 2 introduces basic terminologies and BE-PUM. Section 3 describes Weisfeiler-Lehman Kernel. In Section 4, we discuss our hypothesis about CFGs of unpacking stubs. Section 5 describes our method for the packer identification and the OEP detection. Then, our experimental results are shown in Section 6. Finally, Section 7 concludes the paper.

Related works. Most of the OEP detection is based on a dynamic analysis. If a packed code is fully executed, the original payload must be somewhere in the memory. `Polyunpack` [6] first applies the static disassembly and then runs dynamic analysis. If an executed instruction is different from disassembly, it is regarded as unpacked. `Omniunpack` [7] observes some sensitive system calls. When such a call is detected, it scans the memory page to find unpacked code. Since the original payload is often unpacked in a newly allocated memory, most of OEP detection tools monitors and/or hooks the access to a dirty page (i.e., `write` occurs on a write-protected page) by setting a write protection on a newly allocated area. OllyBonE[1] is a plugin of *Ollydbg*, and tries to stop when the control jumps to a newly allocated region. It prepares a Windows kernel driver for the page protection of a specified region, and set a target memory area and an exception *break-on-execute*. When the control flow moves to the address inside the protected area, it is regarded as the OEP. However, it fails when packers use anti-debugging with the API `IsDebuggerPresent@kernel32.dll`.

The candidates of the OEP are explored either on-the-fly or ahead-of-time. Renovo [8] is built on the top of an emulation environment, TEMU[2]. It stores

[1] http://www.joestewart.org/ollybone
[2] `http://bitblaze.cs.berkeley.edu/temu.html`

a shadow copy of the memory space of the target file and monitors runtime updates. Alternatively, the entry of recently generated code and data is regarded as the OEP. `QuickUnpack`[3] set the OEP breakpoints, following to a common strategy, e.g., pushing breakpoints at `jump` instruction and inspecting near `popa`. `QuickUnpack` chooses the last trigger to OEP breakpoints as the OEP.

The ahead-of-time search of the candidates for the OEP first prepares the candidate list and check it. In [9], OEP candidates are collected at the page faults. They are checked by the entropy (since an encrypted code has larger entropy, i.e., more random) and the number of API calls placed in the memory. `PinDemonium` [10] detects OEP candidates whether the memory dump (using Scylla [11]) can reconstruct the library function table. `Junstin` [12] is often used to collect the OEP candidates. It monitors the control flow and selects an OEP candidate when jumps to a dirty page. It reduces the candidates by heuristics, e.g., *Unpacker Memory Avoidance* (avoid unlikely pages containing unpacked code), *Stack Pointer Check* (check the stack pointer whether the same as the start of the execution), and *Command-line Argument Access* (check whether the command-line argument is put to the stack at the dirty page access). [13] tried to reduce the OEP candidates by identifying decryption routines (by watching writing instructions and written areas) and sorts the candidates. [14] follows [13], and further watching branching instructions to identify decryption routines and tracking system parameters related to the main function. Then, the nearest to the structured exception handler installation (which is located at the last `write` on `fs:[0]` during the system startup) is regarded as the OEP. Apart from the dirty page tracing, [15] combines statistical and symbolic signs which is the pair of the entropy and the single instruction (either `JMP, JCC, CALL, or RET`).

The graph similarity is applied for malware detection and analyses [16, 17]. After obtaining CFGs of packed codes by the symbolic execution `Angr`[4], the former uses CNN on CFGs and the latter uses the (1-dimensional) Weisfeiler-Lehman kernel on a call graph. However, they do not care to distinguish the unpacking stubs and the payloads. Apart from them, we apply the symbolic execution `BEPUM` [3] and the graph similarity to classify the targets, instead of their machine learning techniques.

2 Preliminaries

2.1 Terminologies on graph

We denote the concatenation of two strings s_1 and s_2 by $s_1.s_2$. For a directed graph $G = (V, E)$ with $E \subseteq V \times V$ and $v \in V$, let

$$ancestors(v) = \{u \mid (u, v) \in E\} \qquad successors(v) = \{u \mid (v, u) \in E\}.$$
$$N(v) = (ancestors(v) \cup successors(v)) \setminus \{v\}$$

[3] `https://www.aldeid.com/wiki/QuickUnpack`
[4] https://angr.io/

The *indegree* $\deg^-(v)$ and *outdegree* $\deg^+(v)$ of $v \in V$ is $|ancestors(v)|$ and $|successors(v)|$, respectively. $v \in V$ is a *source node* (resp. *sink node*) if $\deg^-(v) = 0$ (resp. $\deg^+(v) = 0$). We also sometimes denote $u \to v$ if $(u, v) \in E$. A directed graph G is *acyclic* if there are no $v \in V$ with a cycle $v \to^+ v$.

Assuming a DFS on G, we have the order among children nodes. We say,

- Forward edge: from an ancestor to a direct descendant.
- Cross edge: from a righter node to a lefter node.
- Retreating edge: from a descendant to an ancestor.
- Back edge: Retreating edges (u, v) such that v dominates u, i.e., every path from the roof of the DFS tree to u traverses v.

Definition 1. *Let $G = (V, E)$ be a directed acyclic graph. For $u \in V$, the* predecessor graph *from u is a graph $Pre_u^G = (V_u, E_u)$ with*

$$V_u = \{v \in V \mid v \xrightarrow{*} u\} \qquad E_u = E \cap (V_u \times V_u) \tag{1}$$

If G is clear from the context, we may omit it as Pre_u.

The label of $G = (V, E)$ is a labelling function $l_G : V \to \Sigma$. When $l_G(u) = \sigma$, $\sigma \in \Sigma$ is the label of $u \in V$.

2.2 De-obfuscation for CFG generation

Obfuscation techniques. The difficulty of analyzing packed malware comes from the use of obfuscation techniques introduced by a packer. When a program is obfuscated (e.g., encryption), it may not be interpreted statically. Hence, these packed codes can evade firewall and antivirus scanners. Typical obfuscation techniques are classified into 14 [18], which are further classified into 6 groups [19].

1. **Entry/code placing obfuscation** (Code layout): overlapping functions, overlapping blocks, and code chunking.
2. **Self-modification code** (Dynamic code): overwriting and packing/unpacking.
3. **Instruction obfuscation**: Indirect jump.
4. **Anti-tracing**: SEH (structural exception handler) and 2API (the use of special APIs, `LoadLibrary` and `GetProcAddress` in `kernel32.dll`).
5. **Arithmetic operation**: Obfuscated constants and checksumming.
6. **Anti-tampering**: Timing check, anti-debugging, anti-rewriting, and hardware breakpoints. Anti-rewriting consists of stolen bytes and checksumming.

Among them, Anti-tampering contains VM-awareness and trigger-based behavior, which often affects on dynamic analysis and monitoring.

- Anti-Debugging: It detects the presence of a debug mode by specific API calls, e.g.,*CALL kernel32.IsDebuggerPresent*
- Stolen bytes: This calls *VirtualAlloc* to allocate a buffer, and the unpacked code is written on this area.
- Timing Check: This checks timing anomaly compared to the native Windows environment.
- Hardware breakpoint: Jump destination is stored in debug registers, such as `DR0, DR1, DR2`, and `DR3`.

BE-PUM for CFG generation DSE is considered to be the most powerful tool for de-obfuscation [2, 19]. For instance, DSE overcomes anti-tampering techniques since it can generate satisfiable test instances as long as it is on an executable path. Alternatively, DSE will not explore dead code (e.g., *opaque predicate*) since the constraint for it is detected unsatisfiable.

BE-PUM (Binary Emulation for PUshdown Model)[5] [3] is a DSE tool for binary code on Intel x86/Win32 architecture, which generates the precise CFG of a binary code (including malware). Overall, the architecture of BE-PUM can be illustrated by three main components, which are a CFG storage, a binary emulator, and a symbolic execution (Fig. 2). BE-PUM also adopts JackStab 0.8.3 [20] for one-step disassembly (i.e., interpret one instruction from a given address), Z3 4.3 [21] for a constraint solver (with the *bitvector* backend theory).

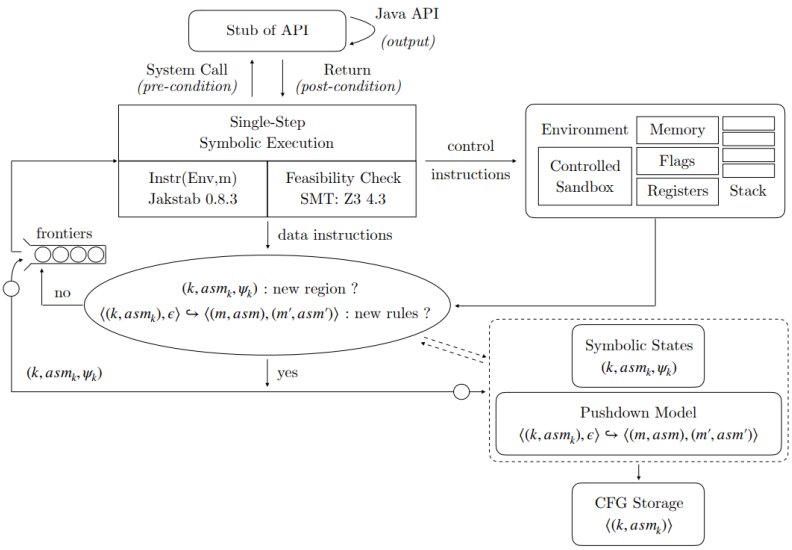

Fig. 2: The architecture of BE-PUM [19]

The frontiers in the left-hand side of Fig 2 store symbolic states (described by the path constraints on symbolic values) at the leaves of currently explored traces. BE-PUM selects one from it and tries to apply the one-step symbolic execution. If it is an indirect jump, one-step testing by a satisfiable instance decides the next destination before the application of symbolic execution. It extends one step of the possible traces, and the path constraint is expanded. This procedure terminates when either the exploration has converged or comes to unknown instructions, unknown system calls, or unknown addresses. Currently, BE-PUM supports about 400 x86 instructions and >1000 Win32 APIs.

[5] https://github.com/NMHai/BE-PUM

3 Weisfeiler-Lehman Kernel

In order to deal with the graph isomorphism, the 1-*dimensional Weisfeiler-Lehman test* has been introduced [4]. It performs with multiple iterations to relabel nodes by compressing the current node labels with the concatenation of the sorted string of node labels of neighboring nodes.

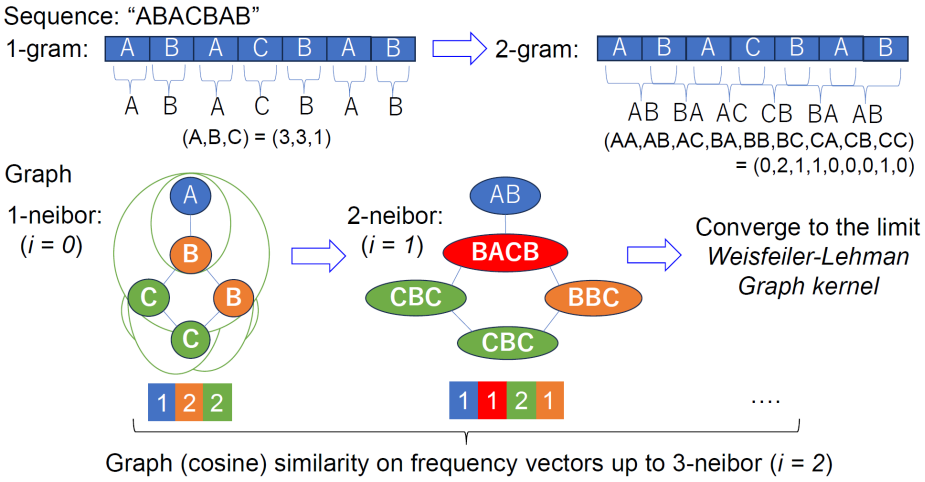

Fig. 3: Weisfeiler-Lehman algorithm

Let $G = (V, E, \ell)$ be a labelled graph with the labelling function $\ell : V \to \Sigma$. We assume that Σ is a totally ordered set. Let $l_0 = \ell$ and $i > 0$. $l_i : V \to \Sigma^*$ is the labeling function after the i-th iteration, which is computed by the following steps. Let $v \in V$ and we assume a total order on Σ.

- **Step 1**: $M_i : V \to \mathcal{M}(\Sigma^*)$ is a multiset-labeling function such that $M_i(v) = \{l_{i-1}(u) \mid u \in N(v)\}$.
- **Step 2**: $s_i : V \to \Sigma^*$ is defined by
  1. $M_i(v)$ is sorted in the ascending order (wrt the lexicographic extension) as $M_i(v) = (l_{i-1}(u_1), l_{i-1}(u_2), \cdots, l_{i-k}(u_k))$.
  2. $s_i(v) = l_{i-1}(v).l_{i-1}(u_1).l_{i-1}(u_2). \cdots .l_{i-k}(u_k)$.
- **Step 3**: $l_i := s_i$.

Fig. 3 shows an analogy of the computation of the Weisfeiler-Lehman Kernel of a graph with the n-gram of a word. n-gram collects the labels of the sequence of the length n. At the i-step, the Weisfeiler-Lehman histogram vector inductively computes the collection of labels in the diameter i of each node. The Weisfeiler-Lehman Kernel continues to compute until the label converges, whereas we use an approximation up to $i = 2$.

For the graph isomorphism, [4] further performs the *label compression* by defining a partial function $f_i : \Sigma^* \to \Sigma$. Adding to the steps above, f_i is incrementally defined starting from $f_0(\sigma) = \sigma$ for $\sigma \in \Sigma$.

– **Step 2'**: For $w = s_i(v)$,

$$f_i(w) = \begin{cases} f_{i-1}(w) & \text{if } w \in Dom(f_{i-1}) \\ v' & \text{otherwise, } v' \text{ is a fresh label added to } \Sigma \end{cases}$$

– **Step 3'**: $l_i := f_i \circ s_i$.

With the label compression, the labeling function l_1 will converge, i.e., $\exists j > 0. l_{j+1} = l_j$. Then, for two labeled graph $G = (V, E)$, $G' = (V', E')$, after both of the labeling functions l_i, l'_i converge at $i = j$, we can simply compare the histogram vectors of $\{l_j(v) \mid v \in V\}$ and $\{l'_j(v) \mid v \in V'\}$. If and only if they are equal, G and G' are isomorphic.

For the CFG similarity, we simplify to directed acyclic graphs after removing retreating edges. Then, we also simplify the algorithm with

– **No label compression**: We use only **Step 1** and **Step 2** and the histogram of $\{l_i(v) \mid v \in V, i \leq j\}$ at the iteration j is called a j-th *Weisfeiler-Lehman histogram vector*.
– **Ancestors instead of neighbors**: In **Step 1**, $N(v)$ instead of $ancestors(v)$.

4 Control flow graph of unpacking stub

4.1 The CFG of the unpacking stub characterizes a packer

We start with our observation that a packed code has a similar unpacking stub regardless of the original payload. The left hand side of Fig. 4 shows the comparison of two CFGs of the packed code by UPX. From this observation, we further set the hypothesis that a similar class of CFGs of unpacking stubs does not cross different packers. If it works, we can identify the used packer and the unpacking stub by the graph matching. (We confirm it by our preliminary experiments in Section 6.) Note that the graph matching will be not exact, since even if the CFGs are in the same class, they may have different offsets, which make binary codes different. To remedy this, we apply the graph similarity, i.e., the cosine similarity [5] on Weisfeiler-Lehman histogram vectors, after annealing the labels by stripping arguments from the instructions.

When we know both the unpacking stub and the original payload, we can identify the body of the unpacking stub as the difference between the memory image after the execution of the packed code and the original payload. Hence, in theory, the CFG of unpacking stubs will be the predecessor graph at the exit of the unpacking stub. However, we sometimes observe a path from the unpacked payload to the unpacking stub. This looks strange since the original payload does not know the unpacking stub in advance. We observe that this happens when the unpacking stub and the unpacked payload call the same API (Fig 5),

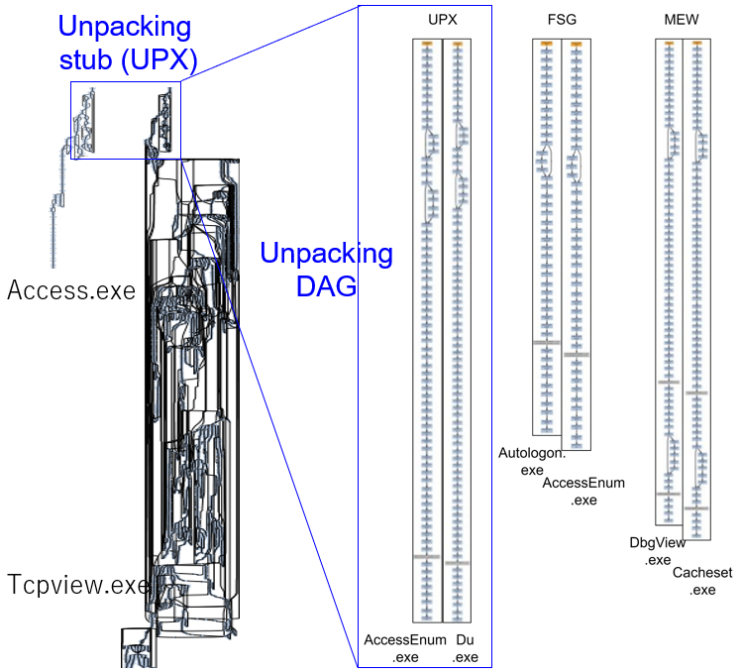

Fig. 4: The similarly among predecessor graphs in the same packer

and this is because the CFG generated by BE-PUM is context-insensitive, i.e., there are no criteria to distinguish different call-sites.

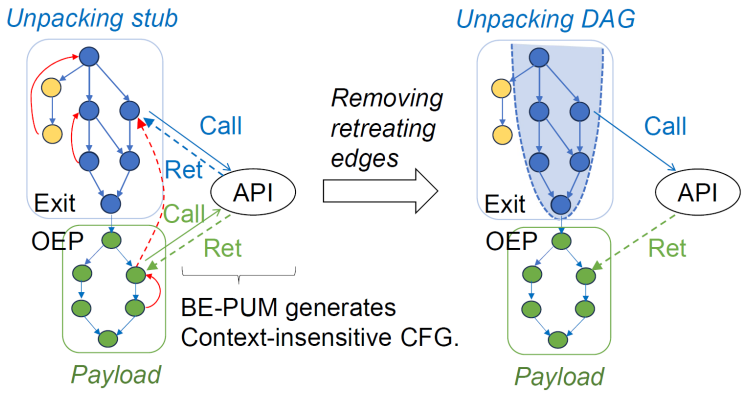

Fig. 5: Unpacking DAG construction

To remedy the situation, we simply strip the retreating edges in the CFG, and then taking the preceding graph at the exit of the unpacking stub. We call it the *unpacking DAG*. Note that the unpacking DAG avoids a loop in the unpacking stub (e.g., a decrypting loop), which may reduce the size of the unpacking DAG much smaller than the original CFG. The right-hand side of Fig. 4 shows the unpacking DAG examples of UPX, FSG, and Mew. They look similar.

As the last remark, a retreating edge in a directed graph depends on the order of nodes travelled in the DFS. However, CFGs of most programs are reducible [22, 23], hence retreating edges in this graph are also back-edges and uniquely selected. In case not reducible, we prepare the DFS strategy that prioritizes visiting a child node with an API call, which intends to remove a return edge from a shared API call both from the payload and the unpacking stub.

End sequence Although these predecessor graphs at the exit of unpacking stubs are clearly classified, they still are not identical. To compensate for the approximation, we observe that the end sequence of these graphs in the same class is consistent. Here, the end sequence is the opcode sequence of the last m instructions in these graphs. Currently, we set $m = 5$. The table below shows several end sequences (proceeding left-to-right) for UPX, FSG, and MEW.

Packer	**the end sequence of unpacking stub**
UPX	`pushl, cmpl, jne, subl, jmp`
FSG	`je, decb, jne, decb, je`
MEW	`GETPROCADDRESS-KERNEL32-DLL, stosl, testl, jne, ret`

The end sequence of unpacking stubs has an important role in our method because Weisfeiler-Lehman kernel is a statistical method and it just approximates a graph into a vector. However, an end sequence is a symbolic property. Therefore, it can reduce the number of unpacking stub candidates and confirm again whether a node in a graph is the exit of an unpacking stub.

4.2 Confirming hypothesis by clustering packed codes

After intuitive observation, we confirm our hypothesis by clustering 771 samples packed by 12 packers (UPX v3.95, ASPACK v2.12, FSG v1.0, YODA v1.3, MEW SE v1.2, PACKMAN v1.0, PECOMPACT v2.xx, PETITE v2.1, WINUPACK v0.39 Final, JDPACK v1.01, MPRESS v2.xx, and TELOCK v0.98). After taking Weisfeiler-Lehman histogram vectors of the CFGs of unpacking stubs (since we know both the original payload and the used packer), the clustering algorithm DBSCAN with the cosine similarity is applied. The table shows the results of DBSCAN with fixed parameters $min\_sample = 2$ and $metric = cosine$. We observe the change wrt *eps* from 0.1 to 0.02.

eps	0.1	0.09	0.08	0.07	0.06	0.05	0.04	0.03	0.02
number of class	11	11	12	14	16	16	18	18	21
Hypothesis (1)	False	False	False	False	False	False	True	True	True
Hypothesis (2)	False	False	False	False	False	False	False	False	True

We observe that when the allowance *eps* of DBSCAN is small enough,

(1) each class does not cross different packers, and
(2) the end instruction sequence (the prefix of the exit of an unpacking stub of the length 5) is the same in each class.

More precisely, $eps = 0.04$ is enough for (1), and $eps = 0.02$ satisfies both.

5 Packer Identification end OEP detection using template matching

In Section 4, we observe that Weisfeiler-Lehman histogram vectors and the end sequence characterize each class of unpacking stubs. We define a *template* of a packer by the pair of the average of Weisfeiler-Lehman histogram vectors and the end sequence. Note that some packers may have several templates.

5.1 Template setup for each packer

Clustering procedure The process of template setup for a fixed packer includes 5 steps. We use the CFG of a packed code generated by BE-PUM.

- **Step 1**: Generate the unpacking DAGs of the unpacking stub.
- **Step 2**: Compute their Weisfeiler-Lehman histogram vector.
- **Step 3**: Apply the 0-aligning, i.e., fulfilling 0 to make the dimensions of vectors the same.
- **Step 4**: Apply the clustering algorithm DBSCAN with the cosine similarity.
- **Step 5**: Pair the average of Weisfeiler-Lehman histogram vectors in each cluster and the end sequence (if it is consistent).

At **Step 4**, we apply DBSCAN with $eps = 0.05$. However, some clusters have inconsistent end sequences. In such a case, we decrease the *eps* value by the step 0.01 and perform again the clustering until stabilized.

5.2 Template matching for packer identification and OEP detection

When we face an unknown packed code, the template-matching process consists of 4 steps. We use the CFG of a packed code generated by BE-PUM.

- **Step 1**: During incremental DFS trace of the CFG, remove a retreating edge and generate the predecessor graph at each node.
- **Step 2**: Compute its Weisfeiler-Lehman histogram vector.
- **Step 3**: If the end sequence of the predecessor graph matches that in a template, check the similarity between the Weisfeiler-Lehman histogram vectors.
- **Step 4**: Choose the template of the maximum similarity.

The node at **Step 4** (the sink node of the predecessor graph) is recognized as the exit of the unpacking stub, and the OEP is detected as the jump destination from it. The packer is identified simultaneously. Fig. 6 shows an example of steps 3 and 4, in which the node 7 is the exit of the unpacking stub, the node 8 is the OEP, and P is the used packer.

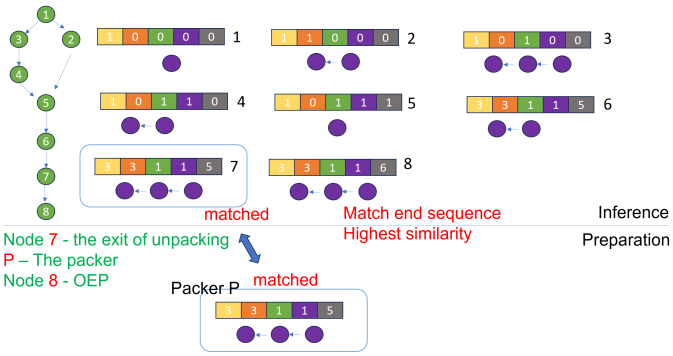

Fig. 6: Template matching for Packer Identification and OEP detection

6 Experiment

A packed code with the used packer name and the original payload is taken from Git Hub pages[6][7] (the latter for TELOCK). We prepare CFGs of packed code by BE-PUM with a time limit of 1 hour, which results in 771 samples. We also prepared 1259 samples of malware from VXHeaven. Windows environment of BE-PUM is set to Windows 7-32bit except for TELOCK. Telock requires `sysenter`, which currently works in BE-PUM only with Windows XP 32-bit. The experimental environments are built on VMware Workstation Pro 17 with Host OS Ubuntu 20.04 and Processor 13th Gen Intel(R) Core(TM) i9-13900K 3.00 GHz. Details of experiments are shown at the GitHub Link[8].

6.1 Testing on non-malware samples

In this experiment, we used 71 samples for obtaining templates, and 700 samples for testing. Our proposed method is compared with

- **Packer identification:** `VirusTotal` (1), `PyPackerDetect`[9](2), BE-PUM [19] (which finds the used packer by the frequency of obfuscation techniques).
- **OEP detection:** `Gunpacker v0.5`[10](3), `QuickUnpack v2.2`[11](4).

Among OEP detection tools mentioned in **Related Work**, we could not find access to most of them [7–9, 13–15], except for `Polyunpack` [6] source[12].

Roughly speaking, the execution time depends on the number of nodes in the CFG. Our method takes mostly around 6 sec for 550 nodes, and 260 sec for 3500 nodes, respectively, whereas `GunUnpacker` and `QuickUnpack` take 1-2 sec.

[6] https://github.com/chesvectain/PackingData
[7] https://github.com/packing-box/dataset-packed-pe
[8] https://github.com/hungpthanh/oep-detection-based-on-graph-similarity
[9] https://github.com/cylance/PyPackerDetect
[10] https://webscene.ir/tools/show/GUnPacker-v0.5
[11] https://www.aldeid.com/wiki/QuickUnpack
[12] https://github.com/PlatonovIvan/PolyUnpack

Packer	Samples	Packer Identification				OEP detection		
		(1)	(2)	BE-PUM	Ours	(3)	(4)	Ours
UPX v3.95	85	85	30	84	85	78	85	85
ASPACK v2.12	68	68	68	68	68	56	68	68
FSG v1.0	75	75	75	75	75	70	75	75
PECOMPACT v2.xx	27	27	27	27	27	0	8	27
MEW SE v1.2	75	75	75	75	75	74	8	75
YODA's Cryptor v1.3	74	74	74	62	74	73	8	74
PETITE v2.1	34	34	34	34	34	0	8	34
WINUPACK v.039 final	26	26	26	26	15	26	4	15
MPRESS v2.xx	78	78	0	78	78	0	8	78
PACKMAN v1.0	79	79	79	0	79	79	8	78
JDPACK v1.01	52	51	0	0	52	45	2	52
TELOCK v0.98	27	27	27	27	27	24	1	27
Total	700	699	515	556	689	525	283	688

6.2 Observation on the result of experiments

For the packer identification, our method correctly detects the packer for 689, next to `VirusTotal` (a database collected from various resources) for 699.

- The failure of `PyPackerDetect` and BE-PUM would come from the obsolete setups, i.e., the supported versions of packers seem too old. The packer identification of BE-PUM is in built-in service [19], which compares the frequency of the occurrences of obfuscation techniques, called *metadata signature*. BE-PUM also detects whether packed by the presence of overwrite the code (e.g., self-modification, encryption). However, the metadata signatures are not updated (which causes failures for UPX and YODA), and BE-PUM does not support PACKMAN and JDPACK. Similarly, `PyPackerDetect` fails on MPRESS and JDPACK.
- Our method for the packer identification fails on 11 samples by WINUPACK, which report "*unknown packer*". The reason is, although WINUPACK has at least 2 templates (Fig 7), our method found only the left one due to a small number of samples (the total 71 for 12 packers) for the template setup.

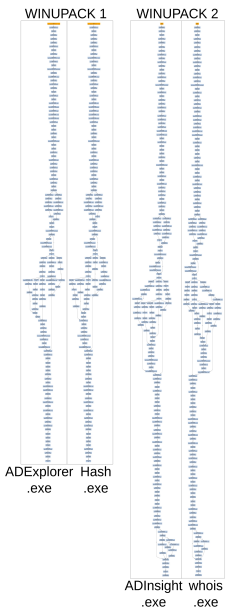

Fig. 7: WINUPACK templates

For the OEP detection, our method correctly detects the OEP for 688, whereas `GunUnpacker` and `QuickUnpack` find 525 and 283, respectively.

– Our method fails 11 samples in WINUPACK (of which the packer identification already fails) and 1 sample in PACKMAN. This shows that once the used packer is correctly identified, our OEP detection is mostly correct. Fig. 8 shows the irregular behavior of the failed sample of PACKMAN. We observed that there is an instruction `jmp start`, which strangely repeats the unpacking process one more time before jumping to the OEP.

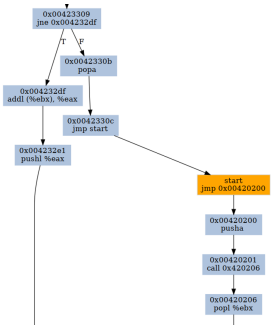

Fig. 8: PACKMAN failure

– The table below shows the metadata signature of BE-PUM [19], in which the number in the top column indicates the categories of obfuscation techniques [18], e.g., (3) overwriting and (4) Packing-unpacking. `Gunpacker` fails on PECOMPACT, PETITE, and MPRESS, which seem to have significantly more (3) overwriting and (4) Packing-unpacking obfuscations.

Packer	Average frequency of obfuscation techniques													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
UPX	0	2.27	0.53	0.1	16.14	17.41	0.41	11	5.57	0.28	0.16	0.03	0.13	0.13
ASPACK	2	0	0	0.5	76.73	8.97	0	1.5	12.92	0	0	0	2	0
FSG	0	0.09	1.54	0.13	14.27	27.34	0.28	16.05	3.74	0	0.16	0.03	0.14	0.1
PECOMPACT	0.53	0.61	0.7	**1.12**	**106.56**	18.8	1.06	6.89	20.63	0	0.04	0	3.03	1.01
MEW	0.3	0	2.65	0.03	55.84	27.66	0.29	14.38	16.67	0.01	0.13	0.02	0.15	0.1
YODA	0	1.77	5.08	0.14	17.15	25.12	1.11	16.75	13.84	1.01	0.17	0.89	0.14	0.96
PETITE	0	0.03	1.65	**3.46**	**180.79**	9.31	2.4	1.81	26.44	0.05	0.11	0	0.25	0.12
WINUPACK	0.47	0.47	0.1	0.09	21.12	22.91	0.07	2.24	4.67	0.76	0.04	0	0.01	0.03
MPRESS	0	0	1.94	**1.88**	**93.81**	0.94	0	0.94	9.82	0	0	0	0	0
PACKMAN	0.48	0	0	0.02	6.87	4.8	0	0.94	0.97	0	0	0	0.99	0
JDPACK	3.09	5.59	0.33	0.07	19.32	27.16	0.16	16.62	2.81	0	0.23	0.01	1.07	0.05
TELOCK	0	2.94	3.96	0	37.42	7.56	9.81	0.21	11.33	0	0.02	0	0	1.98

6.3 Experiment on real malware

We also tried 5190 malware samples taken from VXHeaven. Among them, BE-PUM generates CFGs for 1239 within a 1-hour timeout.

From them, our method detects 1089 packed by *unknown packers* (i.e., beyond the prepared templates of 11 packers) and 150 are identified the used packer names (among 11 packers except for Telock). For them, the OEPs are also reported. The table shows the identified packers.

Packer	UPX	ASPACK	FSG	PECOMPACT	YODA	WINUPACK	MPRESS
Sample	80	26	1	9	13	20	1

When a packer is identified, our method also detects the OEP, though currently, we cannot verify its correctness. However, the end sequence of the unpacking stub would rarely match by luck, and their similarity to our template is higher than 0.51, among them, 133 samples have a similarity of more than 0.7.

7 Conclusion

This paper proposed a method for the packer identification and the OEP (Original Entry Point) detection based on the graph similarity of CFGs (Control Flow Graphs) of packed codes. The CFG generation of packed code owes fully on a DSE tool BE-PUM [3] on x86-32/Windows. Our tool for the graph similarity is a *Weisfeiler-Lehman histogram vector* (Section 3) of a CFG, which is computed iteratively by relabeling each node with the collection of neibor's labels [4].

First, we confirmed the hypothesis that the CFG of the unpacking stub characterizes a packer by the clustering of 771 samples packed by 12 packers (UPX, ASPACK, FSG, Yoda, Mew, Packman, PECOMPACT, Petite, WINUPACK, JDPACK, MPRESS, and TELOCK). Second, we set the template, which is the pair of the average of *Weisfeiler-Lehman histogram vectors* and the end sequence. Experiments showed the effectiveness of our method, especially on the OEP (Original Entry Point) detection. For 700 non-malware samples (which are packed by 12 packers above), our method correctly detected the packer for 689, and the OEP for 688. For 1239 malware samples, 1089 were detected packed by *unknown packer* and 150 were packed by the 11 packers (except for TELOCK). Throughout the experiments, when the packer identification succeeds, the OEP seems to be correctly detected (except for 1 case packed by PACKMAN). The impact on the OEP detection is much more substantial than on packer identification, since analyzing the original payload would give new sights on malware.

Future work In the future, we would like to extend our methods to many other packers. Particularly, we also extend our methods to packers using vm-protect techniques (i.e., themida). In addition, the limitation of our current work is that we need to access the packer's executable to generate the templates. We hope to tackle custom packers, of which executables are not accessible.

Acknowledgement

This research is partially supported by JSPS KAKENHI 20K20625 (Grant-in-Aid for Challenging Research).

References

1. J.C.King. "Symbolic Execution and Program Testing", CACM, 19, 385-394, 1976.
2. J.Salwan, S.Bardin, M.-L.Potet. "Symbolic Deobfuscation: From Virtualized Code Back to the Original," DIMVA, LNCS 10885, 372–392, 2018.
3. N.M.Hai, M.Ogawa, Q.T.Tho. "Obfuscation Code Localization Based on CFG Generation of Malware", FPS, LNCS 9482, 229-247, 2015.
4. N.Shervashidze, P.Schweitzer, E.J.van Leeuwen, K.Mehlhorn, K.M.Borgwardt. "Weisfeiler-Lehman Graph Kernels", J. Mach. Learn. Res. 12, 2539-2561, 2011.
5. Wikipedia."Cosine similarity," `https://en.wikipedia.org/wiki/Cosine_similarity`.
6. P.Royal, M.Halpin, D.Dagon, R.Edmonds, W.Lee. "PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware", ACSAC, 289-300, 2006.
7. L.Martignoni, M.Christodorescu, S.Jha, "OmniUnpack: Fast, Generic, and Safe Unpacking of Malware", ACSAC, 431-441, 2007.
8. M.Kang, P.Poosankam, H.Yin. "Renovo: A hidden code extractor for packed executables", WORM'07, 46-53, 2007.
9. R.Isawa, M.Kamizono, D.Inoue, "Generic Unpacking Method Based on Detecting Original Entry Point", NIP, LNCS 8226, 593-600, 2013.
10. S.D'Alessio, S.Mariani. "PinDemonium: a DBI-based generic unpacker for Windows executables", BlackHat, 1-56, 2016.
11. NtQuery."Scylla - x64/x86 imports reconstruction," `https://github.com/NtQuery/Scylla`.
12. F.Guo, P.Ferrie, T.C.Chiueh. "A Study of the Packer Problem and Its Solutions", RAID, LNCS 5230, 98-115, 2008.
13. R.Isawa, D.Inous, K.Nakao. "An original entry point detection method with candidate-sorting for more effective generic unpacking", IEICE Trans. E98-D(4), 883-893, 2015.
14. G.M.Kim, J.Park, Y.H.Jang, Y.Park. "Efficient Automatic Original Entry Point Detection", Journal of Information Science and Engineering, 35, 887-901, 2019.
15. G.Jeong, E.Choo, J.Lee, M.Bat-Erdene, H.Lee. "Generic unpacking using entropy analysis", MALWARE, 98-105, 2010
16. A.V.Phan, L.M.Nguyen, H.Y.L.Nguyen, L.T.Bui. "DGCNN: A convolutional neural network over large-scale labeled graphs", Neural Networks 108, 533-543, 2018.
17. C.-H.B.Van Ouytsel, A.Legay. "Malware Analysis with Symbolic Execution and Graph Kernel," NordSec, LNCS 13700, 292–310, 2022.
18. K.A.Roundy,B.P.Miller. "Binary-code obfuscations in prevalent packer tools," ACM Comput.Surv.46, 4:1–4:32, 2013.
19. M.H.Nguyen, M.Ogawa, Q.T.Tho. "Packer Identification Based on Metadata Signature", SSPREW-7, 1-11, 2017.
20. J.Kinder, F.Zuleger, H.Veith. "An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries", VMCAI, LNCS 5403, 214-228, 2009.
21. L.de Moura, N.Bjørner. "Z3: An Efficient SMT Solver", TACAS, LNCS 4963, 337-340, 2008.
22. D.E. Knuth. "An empirical study of FORTRAN programs", Software Practice and Experience, 1(2), 105–134, 1971.
23. M.S.Hecht, J.D.Ullman."Flow Graph Reducibility", ACM STOC, 238-250, 1972.