Birds of a *Feature:* Intrafamily Clustering for Version Identification of Packed Malware

Leo Hyun Park, Jungbeen Yu, Hong-Koo Kang, Taejin Lee, and Taekyoung Kwon, *Member, IEEE*

***Abstract*—It is challenging for malware lineage inference to identify versions of collected malware by ensuring high accuracy in clustering. In this article, we tackle this problem and present a novel mechanism using behavioral features for *version identification* of (un)packed malware. Our basic idea involves focusing on intrafamily clustering. We extract the so-called family feature sets, i.e., hybrid features specific to each family. Our intuition is that family feature sets may achieve higher accuracy in clustering than common feature sets, and unpacked malware found in or relevant to such a cluster can result in the lineage inference of family members using traditional inference methods. We conduct experiments with two datasets, 8928 malware samples from *VXHeavens* and 3293 samples by manual analysis, composed of packed malware in a large portion. The results demonstrate that we can accurately classify samples into malware families based on the hybrid features we choose. In addition, we can also effectively extract family feature sets from 37 feature categories using forward stepwise selection. For intrafamily clustering, we employed the agglomerative clustering algorithm and observed that using family feature sets is significantly more accurate than using common feature sets, which facilitates higher accuracy lineage inference of packed malware.**

***Index Terms*—Family classification, feature selection, lineage inference, malware, packing, within-family clustering.**

I. Introduction

THERE is a substantial growth in the amount of malware emerging annually. According to *AV-TEST*, the number of malware samples reported in 2008 was approximately 10 million, which increased to 127 million in 2015, indicating a 12-fold increase [4]. However, most malware are variants of existing ones. According to *G DATA Software*, new malware species comprise only 7.41 million among 110 million malware samples identified in 2017 [15], and it is estimated that *over 80% of malware* are "packed" on distribution [23].

Manuscript received June 25, 2019; revised November 12, 2019; accepted December 6, 2019. Date of publication January 7, 2020; date of current version September 2, 2020. This work was supported in part by the Institute of Information and Communications Technology Planning and Evaluation under Grant 2017-0-00158 (Development of Cyber Threat Intelligence Analysis and Information Sharing Technology for National Cyber Incident Response) funded by the Korea Government (Ministry of Science, ICT and Future Planning) and in part by the Institute for Information and Communications Technology Promotion under Grant 2018-0-00513 (Machine Learning Based Automation of Vulnerability Detection on Unix-Based Kernel) funded by the Korea Government (Ministry of Science, ICT and Future Planning). *(Corresponding author: Taekyoung Kwon.)*

L. H. Park, J. Yu, and T. Kwon are with the Yonsei University, Seoul 03722, South Korea (e-mail: dofi@yonsei.ac.kr; symnoisy@yonsei.ac.kr; taekyoung@yonsei.ac.kr).

H.-K. Kang is with the Korea Internet and Security Agency, Naju 58324, South Korea (e-mail: redball@kisa.or.kr).

T. Lee is with the Hoseo University, Asan 31499, South Korea (e-mail: kinjecs0@gmail.com).

Digital Object Identifier 10.1109/JSYST.2019.2960076

Plenty of studies about malware classification based on machine learning have been performed. This area has been reaching its maturity in both academia and industry. However, even though malware classification can protect an end system, it does not help malware analysts understanding malware ecosystem in depth. Malware lineage inference involves exploring the evolutionary relationships among collected malware samples. Malware variants developed by the same malware authors or variated from the same sources comprise the lineage. Lineage inference plays an important role in malware forensics, such as tracking the provenance of malware and naming malware samples. In particular, lineage inference can help determine which sample must be analyzed first and help evaluate trends in the evolution of malware. Therefore, malware lineage inference is a crucial step to learn how to cope with new malware in arms race between malware authors and analysts. Past lineage inference studies attempted to determine which malware samples had been developed in the past and which samples had been developed relatively more recently [16], [21], [26], [31]. The order of malware samples was denoted as a graph, and the shape of the graph was constructed using features such as file/section sizes, control flow graphs, and function similarities. The graph comprises nodes and edges; nodes indicate each version of a family, and an edge connects two adjacent versions.

Lineage inference is still challenged with the requirement that it is necessary to handle not only unpacked malware, but also packed malware. Unfortunately, prior studies failed to consider packed malware, and they considered all samples in a malware family as different versions. Given N versions of a family, $N * (N+1)/2$ complexity is required to identify their chronological sequence. The complexity problems associated with lineage inference have not been revealed from the evaluation of previous studies that consider a small dataset, i.e., small N. However, if a large amount of packed malware coexist in the same family, the computational complexity for lineage inference may increase explosively. In reality, most samples are packed, which means that the size of N can be dramatically reduced.

In this context, *version identification* is a crucial step for filtering packed malware before performing lineage inference. A prior study categorized packed malware and unpacked malware as the same version. Haq *et al.* [21] unpacked and disassembled a sample and then verified for function similarity with other samples. However, their results were not satisfactory because their function coverage metric was accurate approximately 70% on average. Contrary to their approach, our perspective assumes dynamic analysis as inevitable for accurate grouping of unpacked malware and packed malware characterized by the same behavior.

In this article, we present a new method of version identification to mitigate the complexity problem associated with

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see http://creativecommons.org/licenses/by/4.0/

lineage inference involving a large amount of packed malware. We propose intrafamily clustering for version identification. Intrafamily clustering easily groups a version of packed malware and unpacked malware according to behavioral features that can be extracted through dynamic analysis. Unpacked malware whose lineage has not been identified can be detected after intrafamily clustering, and its lineage can be easily inferred. It should be noted that we do not execute lineage inference itself but refine an input for lineage inference based on version identification.

Our system design is straightforward and executes stepwise *feature processing* and *version identification*. We first extract malware features for machine learning, from sandbox, and for static analysis. In particular, we employ several behavioral features that can be extracted from the *Cuckoo Sandbox* [11]. Features that cannot be immediately employed for machine learning are processed using feature representation. We also reduce the dimensionality of behavioral features to improve memory utilization. Once a feature set of samples is available, the feature sets are classified into families using classification algorithms. We perform *forward stepwise selection* and *intrafamily clustering* for each classified family. We are concerned that feature selection employed in prior studies [1], [14], [16], [20], [40] was commonly used, and some malware families may not have fit into the commonly selected feature set. This is because those families exhibit behaviors that are different from the common characteristics. Our primary assumption is that family features may exist upon which each malware family's member behaves with similar sensitivities, just as birds of the same species exhibit similar behaviors and characteristics. Therefore, we derive the so-called family features, not a common one. Finally, we choose a cluster head in each cluster for lineage inference input. Our entire system, including version identification, was evaluated on a large-scale dataset containing 8928 malware samples obtained from *VXHeavens* and 3293 samples obtained through manual analysis. The primary contributions of our article are summarized as follows.

1) *New version identification system:* We propose an integrated system that includes feature processing, family classification, and intrafamily clustering for malware version identification. Our system identifies a version of packed malware and eventually prevents packed malware from infecting lineage inference. We expect that our system dramatically improves the computational complexity of lineage inference.
2) *Feature processing for accuracy and efficiency:* In addition to static features obtained through binary analysis, we also extract dynamic features through sandbox analysis. Consequently, we can precisely handle packed malware in family classification and intrafamily clustering. In addition, we propose a feature normalization method that reduces the dimensionality and memory requirements of a feature set, which preserves the meaning of behavioral information.
3) *Intrafamily clustering based on family feature sets:* We employ a feature selection method and derive family feature sets based on the behavior and sensitivity of individual families. Family feature sets can improve the accuracy of intrafamily clustering, i.e., version identification. We observed that malware families exhibit different sensitivities, and our family feature sets represent those characteristics well.
4) *Experiments with a large-scale real-world dataset:* We collected malware samples from two sources and constructed a large-scale dataset for evaluating our entire system. Our system classifies malware samples into families and demonstrated F_1-scores of above 98%. Furthermore, intrafamily clustering based on family feature sets results in an F_1-score of about 90%, which indicates a considerable increase from prior version identification studies, e.g., 70% approximately. Our experimental results also demonstrate that our approach improves performance compared to previous approaches in terms of feature processing and feature selection.

The rest of this article is organized as follows. In Section II, we describe a background of this study. (Advanced readers may skip this.) In Section III, we present our system design. We then explain feature processing and version identification in Sections IV and V, respectively. In Section VI, we perform evaluation with thorough experiments. In Section VII, we discuss practical impacts and limitations of our system. We describe related work in Section VIII. Finally, Section IX concludes this article.

II. Background

To understand the intent and behavior of malware samples, family classification must be considered an essential process in malware analysis. Once the family of an unknown sample has been identified, analysis of the sample can be easily performed based on prior knowledge. Moreover, in terms of software engineering, malware authors continuously develop malware for bypass detection mechanisms or add functions. For example, malware authors submit their malware to a public sandbox to fingerprint the environment of the sandbox and then develop more sophisticated malware [16]. Lineage inference is essential to track malware evolution and to identify the relationships among malware versions. However, evasion techniques present difficulties in terms of automating family classification and lineage inference. Packing, one of the most typical techniques, has been applied to more than 80% of malware samples [23]. Packing compresses a binary including a code section and a data section to reduce its size and applies obfuscation to avoid processes such as reverse engineering. Packing outputs a variation of the original malware binary, making it difficult to identify malware based on static features. Using an unpacker tools is one solution for handling packed malware using static features, but it is difficult to determine which unpacker must be applied.

There have been many prior studies that focused on the classification of packed malware based on static, dynamic, and hybrid features; however, most studies did not consider lineage inference [1], [6], [23]. In terms of using lineage inference to trace malware development, several studies have been conducted, but these studies have limitations [16], [21], [26], [31]. It is essential to consider the packing problem in lineage inference as prior studies had difficulties in identifying packed malware. Graziano *et al.* filtered packed samples from their dataset [16]. Lindorfer *et al.* applied simple generic unpacking to handle packed malware, but they were concerned about repacked malware [31]. Jang *et al.* used dynamic application programming interface (API) calls, but their approach was not proven on packed malware [26].

The separation of unpacked malware and packed malware must be considered in lineage inference; otherwise, all samples

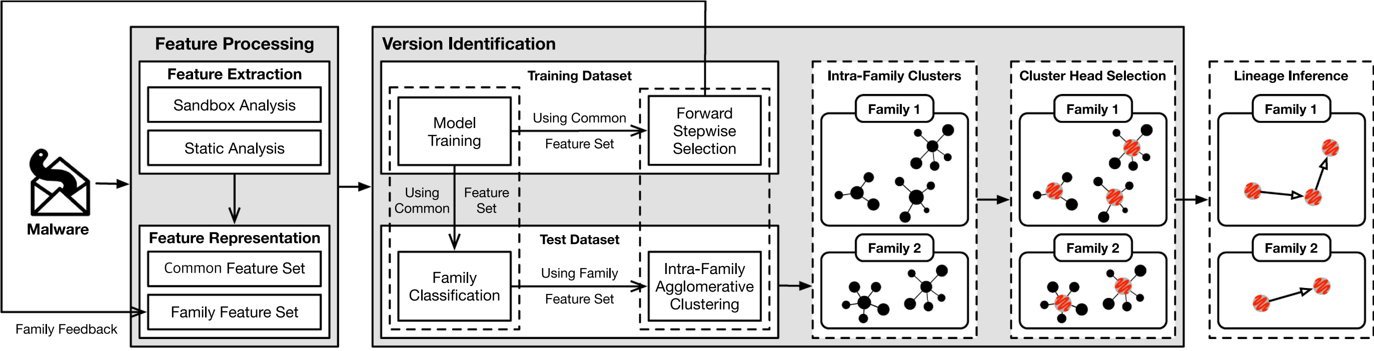

Fig. 1. System overview.

would be compared to each other, which exponentially increases computation costs. Moreover, inferring the chronological sequence of unpacked malware has been well defined, except in the case of packed malware; one example involves a case in which two packed malware samples originate from the same unpacked sample but are packed with different packers. Therefore, lineage inference must be performed on only unpacked samples. In the case of packed samples that originate from the same unpacked sample, the samples must be grouped according to their versions, i.e., version identification. In this context, Haq *et al.* attempted to identify a version of packed samples for efficient lineage inference [21]. They first unpacked and disassembled executables. Their version identification process was based on function similarity of the two samples. However, as the authors employed static features, the performance was relatively low. Furthermore, although repacked samples from the same packer could be handled, samples packed with more than two packers were difficult to analyze. Therefore, behavioral features that can be discovered beyond packers must be utilized for accurate version identification.

III. System Design

A. System Overview

We propose a new method of version identification so that we can create compatible inputs for lineage inference from large-scale malware datasets. Our main objective is to derive so-called family features from static and dynamic behaviors of malware variants and utilize them for more accurate clustering of packed malware. Our system design is straightforward, as illustrated in Fig. 1, and executes stepwise feature processing and version identification.

B. Feature Processing

Malware features must be extracted and represented before version identification. Feature processing consists of two phases: feature extraction and feature representation.

In the *feature extraction* phase, we extract pairs of static and dynamic features from malware samples. In other words, dynamic link library (DLL), API, entropy, section information, etc., are extracted through static analysis, while dynamic API call sequences and other behavioral features (e.g., file, registry, network, and mutex) are extracted through sandbox analysis.

In the *feature representation* phase, we denote textual data, such as an API call sequence or dynamic behavior, as numeric feature vectors. Furthermore, we reduce the dimensionality of behavioral features to reduce memory requirements. In this process, we apply different normalization methods to each category, considering the semantics and forms of each behavior. Our distinguishing point is to separate malware features into two categories: the common feature set and the family feature set for different purposes.

C. Version Identification

In the *family classification* phase, malware samples that underwent sandbox and static analysis are first classified into families before being clustered. We utilize several classification algorithms implemented in *scikit-learn* [39]. We apply the algorithm, which can result in the best performance for *family classification*.

Once the family of a sample has been identified, we identify versions of the samples in each family in the *intrafamily clustering* phase. Prior to clustering, in *forward stepwise selection*, we derive a family feature set using known samples for each family to maximize clustering accuracy. This algorithm selects a category that produces the highest F_1-score among the feature category candidates and adds it to the empty feature set. In a stepwise fashion, the remaining candidates are added in order of the highest score. This algorithm ends when an added feature does not improve accuracy.

For unknown samples, we match packed samples to unpacked samples of the same version based on *agglomerative clustering* [9], [16], [23], [25], [26], [28], [30], [34], [36] with the family feature set. After clustering, each intrafamily cluster denotes each version in a family. In each cluster, an unpacked malware sample can be detected through *cluster head selection* for later lineage inference. Although lineage inference itself is beyond the scope of our study, we expect that only cluster heads i.e., unpacked samples, will be fed into the lineage inference process. As prior lineage inference studies have focused on the correlation of unpacked samples [16], [26], the output of our system can be utilized as inputs to their approaches.

Our approach may seem confusing because we use both classification and clustering algorithms; however, the classification and clustering procedures are fully separated and have different purposes. Classification algorithms are used for family classification by creating a model based on the training dataset, which we refer to as known malware samples in our system. The trained model classifies the test dataset, which we refer to as unknown malware samples in our system. On the other hand, clustering algorithms are used for intrafamily clustering but do not involve creating a model. We do not consider known

samples as the training dataset. We rather use the known samples only for selecting family features. The clustering algorithm only identifies the relationships among unknown samples according to selected family feature sets.

IV. Feature Processing

A. Overview

Feature processing involves the extraction and representation of malware features that must be used for family classification and intrafamily clustering based on machine learning. In the feature extraction phase, we essentially employ dynamic analysis to handle packed malware. However, dynamic analysis exhibits difficulties involving trace dependence and antivirtualization, and therefore, to mitigate these problems and improve accuracy, we also apply static analysis, which produces high-code coverage.

In the feature representation phase, we focus on three factors. First, each malware family may exhibit different sensitivities; therefore, it is necessary to extract not only a common feature set, but also a different feature set from each family to improve the accuracy of intrafamily clustering. In the family classification phase, using family feature sets can be challenging because input malware samples have not been classified. We extract a *common feature set*, applied commonly to all families, for family classification. For intrafamily clustering, we extract a so-called *family feature set*, applied differently to each family, where candidates of a family feature set comprise a common feature set and behavioral features. Second, some behavioral information, such as API call sequences, consists of lists of strings whose length is flexible; therefore, we convert these data into numeric feature vectors that are applicable to machine learning. Third, malware samples include a variety of feature attributes, resulting in the curse of dimensionality. We employ dimensionality reduction to reduce the size of features being converted.

B. Feature Extraction

1) Sandbox Analysis: A sandbox can analyze a batch of malware samples without affecting the host system because it initializes a virtual machine state after analyzing a malware sample. Behavioral features beyond API and system calls that are exposed during execution, such as file, network, and registry features, can be extracted by sandbox analysis. We extract tens of behavioral features from the *Cuckoo Sandbox* [11], such as an API call sequence, and details of the behavioral features that the Cuckoo Sandbox offers are described in Table I. Our behavioral features consist of 12 primary categories. Each feature category includes subcategories used for real feature vector construction. Cuckoo Sandbox also includes static features that are extractable from the portable executable (PE) header, and compile time and section information reported by the Cuckoo Sandbox are used as additional features.

Malware with antivirtualization analyzes its running environment and verifies whether it is on a real or virtual machine. When the malware detects a virtual machine, it ends execution or performs another function. Third-party sandbox developers have offered and shared a hardened version of the Cuckoo Sandbox to mitigate this problem [7]. The hardened sandbox can mitigate antivirtualization because it constructs the same running environment as a real machine, and we applied this version of the Cuckoo Sandbox for reliable sandbox analysis.

TABLE I
Malware Behavior Feature Categories Extracted From Cuckoo Sandbox

Category	Subcategory	Category	Subcategory
Host	host	**Mutex**	mutex
Domain	domainIP domainName	**Service**	serviceCreated serviceStarted
ICMP	icmpSrc icmpDst icmpType	**DNS**	dnsType dnsRequest dnsAnswer
Registry	registryWrite registryRead registryDelete registryAccess	**Accessed Files**	fileWrite fileRead fileDelete fileAccess
UDP	udpTime udpSrcIP	**PCAP**	pcapSorted pcapSHA256
	udpSrcPort udpOffset udpDstIP	**Dropped Files**	droppedSSSDEEP droppedSize droppedName
	udpDstPort	**Command**	command

TABLE II
General API Groups by Malware Group

Malware Activity	API Patterns
Key Logger	FindWindowA, GetMessage, ShowWindow, RegisterHotKey, UnhookWindowsHookEx, SetWindowsHookEx, GetAsyncKeyState
Screen Capture	GetDC, BitBlt, GetWindowDC, CreateCompatibleDC, CreateCompatibleBitmap, SelectObject, WriteFile
Anti-debugging	OutputDebugStringA, OutputDebugStringW, IsDebuggerPresent, CheckRemoteDebuggerPresent
Downloader	URLDownloadToFile, WinExec, ShellExecute
DLL Injection	OpenProcess, VirtualAllocEx, WriteProcessMemory, CreateRemoteThread
Dropper	FindResource, LoadResource, SizeOfResource

2) Static Analysis: The DLL and API information imported into the PE files for binding dynamic libraries denote good features for analyzing the behavioral characteristics of malware. We selected 171 DLLs and 1279 APIs to collect optimal features. The intrinsic behavior of malware is denoted by basic units of behaviors, such as a key logger, DLL injection, and antidebugging. When analyzing the appearance of a basic behavior and using this as a feature, the result can be more precise than using a large number of DLLs and APIs. The major DLLs and APIs affected by malware are described in Table II.

Entropy implies a degree of disorder and increases when applying obfuscation techniques such as packing, which can result in changes in sections of the PE file. Therefore, we can derive features of malware by combining the analysis of the PE file and evaluating the degree of entropy.

C. Feature Representation

1) Common Feature Set: We include static features and dynamic API call sequence(s) in the common feature set. We can immediately use the static features extracted in the previous phase, but feature representation is performed to improve accuracy. In the case of major malicious activities, it is important to consider the completeness of the basic unit of operation as a feature. Therefore, the rate of the appeared API patterns among related API patterns is derived as a feature value. In the case of entropy, malware that applies obfuscation techniques tends to use an unknown section name, thereby creating a difference in the distribution of entropy between known sections and unknown sections.

Sequential data are not applicable to machine learning because their length varies across samples; therefore, to fix the length of API call sequences, we represent those sequences as 2-grams, reflecting their shorter sequence pattern and capturing the basic semantics of the program. An N-gram separates sequential data into patterns of a certain length and then counts the frequency of patterns with a sliding window of size N. We extract all API functions that a malware called during execution from the sandbox and list them sequentially. The sliding window increases the frequency of a 2-gram value corresponding to its position, while the window moves from the beginning of an API sequence to the end.

2) Family Feature Set: We include all behavioral features extracted in addition to the static features and API call sequences included in the common feature set to construct the family feature set. The Cuckoo Sandbox represents most behavioral data as a list of strings. As discussed previously, the behavior of flexible length, such as an API call sequence, is inapplicable to machine learning. To apply this type of behavior to machine learning, behavioral data must be converted into a numeric feature vector. We refer to the method that is applied to an API call sequence, which represents behavior as 1-gram. In other words, each behavior of all malware is represented by each attribute, and the value of each attribute is specified by the number of appearances of each behavior.

Our method exponentially increases the size of features because we add all behaviors of malware as attributes. To improve the performance of machine learning, we reduce the dimensionality of behavioral features by reducing the number of attributes by using only a part of each attribute name, considering the semantics and hierarchy of behaviors. We normalize features using different reduction methods depending on the different semantics and form of each category, as follows. With a 32-bit IP address, we use only the first 16 bits, the high-level value. For example, we use only *192.168* from *192.168.56.25* and discard the last 16 bits, the low-level value. We split domain-type data with a comma and then use the last value. We round off floating point values and convert them into integers. We divide port numbers and file size by 100. We split a registry key value with "/" and then use the first and the last values. We employ only extensions of commands and file names. In particular, malware frequently accesses itself, which corresponds to file behavior. We express this case as *itself*. We use the remaining categories without modifications.

V. Version Identification

A. Overview

In this section, we present a description of a method for identifying the versions of a family based on the family classification phase and intrafamily clustering phase. We consider the classification and clustering for different phases. It is essential to classify malware samples into families in the *family classification* phase in advance to identify versions of each family. Clustering algorithms can assign malware samples into groups, but cannot identify which family a sample belongs to. Therefore, we apply a classification algorithm rather than clustering.

We identify versions of classified samples in terms of a family in the *intrafamily clustering* phase. In other words, we identify the relationships among malware samples in a family. As the amount of packed malware that can be created from an unpacked malware is considerable, it is inefficient to identify the lineage of all samples in one family. Therefore, we focus on grouping unpacked malware and the packed malware derived from the unpacked malware into the same version. Our system should also be able to group samples of a new version. We apply a clustering algorithm that can create a variable number of clusters. It must be noted that we do not observe a lineage graph itself of a family. Nevertheless, clusters we create in this phase can replace the elements, i.e., nodes, of a lineage graph that previous studies have created [16], [21], [26].

B. Family Classification

In the family classification phase, a multiclass classifier is applied to classify samples of a number of families. The classifier requires a common feature set and labels assigned to malware families as training data. We train the classifier using a training dataset; for samples in a testing dataset, the family classifier predicts their families. As the performance of machine learning is dependent on the dataset and features, it is crucial to select an appropriate classification algorithm. *Scikit-learn* [39] is a machine learning framework based on *Python* [35] and provides representative classification and clustering algorithms with open-source libraries. The most typical classification algorithms are *Decision Tree (DT), Random Forest (RF), Support Vector Machine (SVM), Naive Bayes (NB), Gaussian Process (GP)*, and *K Nearest Neighbor (KNN)*. We evaluated these algorithms by applying tenfold cross validation.

C. Agglomerative Clustering

Agglomerative clustering is one of hierarchical clustering algorithms, that begins with clusters consisting of only one sample, and then expands clusters by merging two clusters with the highest similarity. We consider the similarity of two clusters as to be the similarity of features of the samples when there is one sample of each in both clusters. When there are more than two samples of each in both clusters, we consider the similarity of the clusters as the average of the similarities of all features in all samples. This process proceeds until there remains no clusters to be merged or all samples have been merged into one cluster.

We utilized the agglomerative clustering libraries provided by *scikit-learn* [39]. The agglomerative clustering of *scikit-learn* requires, in advance, the number of clusters as an input value. It merges clusters in a bottom-up approach and stops when the number of clusters matches the input value. In this article, we set the number of clusters as the number of versions for each family.

However, the clustering must be based on a threshold to accommodate new versions in the real world. In this case, clusters can be merged only when their similarity is above the threshold. This approach is independent of the number of clusters, i.e., versions, but a number of outliers may arise and the accuracy may drop. Therefore, the threshold must be well defined for flexible and accurate clustering.

D. Forward Stepwise Selection

Considering our feature representation method, we compare the performance of each feature category rather than applying a statistical feature selection method. Among the feature categories, we wish to focus on identifying a combination with the

Algorithm 1: Forward Stepwise Selection.

```
procedure forward stepwise selection
  F_original = {f_1, f_2, f_3, ..., f_n}
  F_optimal = {}
  S_optimal, count = 0
  while count < n do
    S_partial = 0
    for all f ∈ F_original do
      S = agglomerative(F_optimal + f)
      if S > S_partial then
        S_partial ← S
        F_partial ← f
      end if
    end for
    if S_partial > S_optimal then
      S_optimal ← S_partial
      F_original.pop(F_partial)
      F_optimal.push(F_partial)
    else
      break
    end if
    count ← count + 1
  end while
    return F_optimal
end procedure
```

highest F_1-score of for clustering; however, there are countless combinations, and therefore, measuring their F_1-scores is calculation intensive. We use forward stepwise selection to reduce calculations involving measurements and comparisons. Forward stepwise selection is a greedy algorithm and measures improvements in performance while evaluating feature candidates in a stepwise fashion. As similar to the pseudocode described in Algorithm 1, the feature category candidates are added in the order of highest accuracy, and this process ends when adding a feature category does not enhance accuracy. If the number of feature category candidates is n, the maximum computational complexity of forward stepwise selection is $n(n+1)/2$. In other words, the algorithm requires a complexity of $O(n^2)$. On the other hand, the number of calculations grow exponentially when investigating all combinations, which requires a complexity of $O(2^n)$.

E. Cluster Head Selection

After clustering, we select a cluster head in each cluster for lineage inference. Considering the fact that packing increases the entropy of a malware binary, several prior studies judged a sample with the lowest entropy as unpacked [24], [33]. However, there are some samples that exhibit reduced entropy as a result of packing. We observed that through packing, the size of known sections (e.g., ".text," ".bss") decreases and the size of unknown sections increases. In addition, some packed samples have section names (e.g., ".upx," ".vmprotect") that indicate that the sample is obviously packed. We first filter out those samples, but there are some packed samples with only known sections. Rather than measuring entropy, we measure the sum of the raw data size of known sections for each remaining sample and then find identify and consider the sample with the highest value in a cluster as the unpacked sample.

TABLE III
SAMPLES USED IN FAMILY CLASSIFICATION (FROM VXHEAVENS)

Family	#	Family	#	Family	#
Frauder	159	Popwin	702	Viking	163
Rukap	133	Sinowal	195	WinSpy	117
Firu	231	Maran	200	Iespy	111
Podnuha	234	BHOSta	145	FlyStudio	226
CcKrizCry	141	Exchanger	253	Flux	179
TDSS (Backdoor)	203	TDSS (Rootkit)	261	BZub	236
Total	3,889				

TABLE IV
SAMPLES USED IN FRAMEWORK EVALUATION (VXHEAVENS DATASET)

Family	Unpacked	Packed	First Compiled	Last Compiled
Renos	13	390	2006-02-17 21:03	2008-10-24 8:13
Cmjspy	31	930	2000-04-13 12:09	2007-12-04 22:16
Lydra	38	1,140	1992-06-19 22:22	2007-08-15 6:43
VanBot	7	210	1970-01-01 0:00	2008-07-24 18:16
Cakl	17	510	1992-06-19 22:22	2006-06-23 10:20
BO2K	22	660	1999-07-04 2:33	2004-11-24 18:01
IstBar	22	660	2003-06-06 16:46	2006-03-23 18:55
Virut	11	330	1997-01-18 16:40	2008-02-22 8:09
SCKeyLog	14	420	2003-02-15 9:42	2006-03-09 13:37
Horst	4	120	2006-11-29 5:10	2006-12-03 0:24
MoSucker	14	420	2000-01-06 18:22	2009-01-28 14:33
Visel	26	780	2006-04-30 3:54	2008-03-09 14:09
Ayolog	26	780	2003-12-12 19:22	2008-07-06 6:41
Xorer	14	420	2000-12-24 11:59	2008-03-17 13:52
DarkMoon	29	870	1992-06-19 22:22	1992-06-19 22:22
Total	288	8,640		

VI. EVALUATION

All of our evaluations were performed on a system comprising an Intel Core i5-6600 CPU @ 3.30 GHz, 32-GB RAM, and Ubuntu 16.04.2 LTS. We set all parameters, except the number of clusters, for ML algorithms as default.

A. Dataset

Our system was evaluated and calibrated based on three types of datasets. Two of these were obtained from *VXHeavens* [22] and the third was the reliable manually created dataset.

We first collected malware samples from VXHeavens. We then filtered samples that did not match the family label from the Cuckoo Sandbox, which is based on the *VirusTotal* family labels. We separated and used samples for verifying group classification and for that of the entire framework, including intrafamily clustering. Table III presents VXHeavens samples used in group classification verification. We excluded families with less than 100 or more than 1000 samples from this dataset, resulting in 3889 malware samples used in group classification verification.

Furthermore, we selected samples to be packed in the remaining families, and created variants using six popular packers. Table IV presents the VXHeavens dataset with 288 unpacked and 8640 packed malware involving 15 families that were used for framework verification. All of the unpacked samples had unique MD5 hash values. We measured the time of compilation (compile time) of the first compiled sample and that of the last compiled sample for each family. We observed that we can identify versions of some samples according to their compile time for each family, but there are other samples with corrupted

TABLE V
SAMPLES USED IN FRAMEWORK EVALUATION (RELIABLE DATASET)

Family	Unpacked	Packed	First Compiled	Last Compiled
Elkern	4	39	1992-06-19 22:22	1992-06-19 22:22
Domaiq	2	20	2014-07-07 8:47	2014-07-07 8:47
Virut	36	1,051	1984-09-23 19:12	2068-12-23 8:50
Heur_gen	5	87	2015-11-22 21:09	2015-12-29 6:43
Zbot	2	60	2004-10-13 3:27	2004-10-13 3:27
Agent*	20	521	1992-06-19 22:22	2010-07-16 8:55
IRCBot*	20	600	2007-03-07 2:55	2011-08-02 17:58
Mamianune	2	51	2001-07-19 19:30	2004-07-12 19:31
Mabezat	2	60	2004-08-04 6:04	2006-10-23 9:48
Turkojan	2	60	1992-06-19 22:22	1992-06-19 22:22
Hupigon*	20	551	1992-06-19 22:22	2012-05-14 14:36
Upatre	4	74	1997-10-25 21:15	1997-10-25 21:15
Total	119	3,174		

compile times. For example, in the family *VanBot*, five unpacked samples exhibited normal compile times, but two exhibited corrupted compile times. This phenomenon also appeared in other families, *Xorer, Cmjspy, Lydra, MoSucker, BO2K, Cakl,* and *Virut*. In another family, *DarkMoon*, all samples exhibited an identical compile time of "1992-06-19 22:22." Therefore, we consider each version of each unpacked sample because we cannot identify the versions of all samples in a family based on the compile time alone.

We used other sources to verify our framework more reliably. First, we collected malware that has been manually classified by a malware expert (20 malware samples in each family). These families are marked with asterisks in Table V. In addition, we utilized three antivirus vendors, Symantec, Kaspersky, and Ahnlab V3. We used only malware samples whose detected names from three vendors are the same. We also packed these unpacked samples using the six packers. Table V presents a reliable dataset with 119 unpacked and 3174 packed malware in 12 families that were used for the entire framework verification. As in the VXHeavens dataset, all of the unpacked samples in the reliable dataset had unique MD5 hash values. We observed that families such as *Virut, Agent*, and *Hupigon* included samples with corrupted compile times. In addition, all samples in each family of *Elkern, Domaiq, Zbot, Turkojan,* and *Upatre* exhibit identical compile times. Therefore, we apply the same consideration as for the VXHeavens dataset.

B. Packers

The six packers we used to pack malware include *UPX, ASPack, PECompact, PETite, NSPack,* and *VMProtect*. *UPX, PETite,* and *PECompact* can adjust a packing depth from one to nine. That is, we can create nine variants from one unpacked sample. As the depth of the other packers is fixed to one, we can create only one variant from one unpacked malware. The maximum number of possible packed samples from one unpacked sample was 30. *UPX and PETite* can automate batch processing via *python*, but the others are based on GUI, and so batch samples were manually constructed.

C. Family Classification

1) Algorithm Selection: We evaluated classification algorithms by employing a tenfold cross validation in which the process is repeated ten times. We used samples of Table III and compared the average classification accuracy of each algorithm.

TABLE VI
PERFORMANCE COMPARISON FOR EACH CLASSIFICATION ALGORITHM

Score	DT	RF	NB	SVM	KNN	GP
F1-score	0.9887	0.9928	0.8832	0.9209	0.9692	0.9370
AUC	0.9695	0.9918	0.8422	0.9844	0.9839	0.9744
Time(sec)	26.7	11.38	9.75	1410	127.07	86564

The abbreviation of algorithms are mentioned in Section V-B.

Table VI presents the F_1-scores and area under the curve (AUC) for the six classification algorithms. *RF* achieved not only the highest F_1-score and AUC but also relatively short analysis time. Malware features such as API call count are generally discrete, while other algorithms suit to continuous features. On the other hand, RF can examine various feature combinations, so it performs well discovering decision boundary of malware families. In this reason, we chose RF for family classification.

2) Results: We measured the performance of family classification based on the selected algorithm. The F_1-scores of family classification were 99.25% and 98.17% for VXHeavens and the reliable dataset, respectively. Fig. 2 presents the overall performance of family classification based on the receiver operating characteristic (ROC) curve. ROC curves of both datasets covered most of the graph, which means that the classification was performed very precisely. Fig. 3 presents detailed results using the confusion matrix. The accuracies for each family were over 99% except for four families of the VXHeavens dataset and two families of the reliable dataset.

D. Intrafamily Clustering

1) Family Feature Set and Clustering Accuracy: In this experiment, we analyze the results of intrafamily clustering based on several feature sets and agglomerative clustering. For each dataset, we used half of the samples, i.e., known samples, for feature selection and then selected features for clustering the remaining half of the samples, i.e., unknown samples.

Tables VII and VIII present derived family feature sets and the common feature set on the VXHeavens dataset and the reliable dataset, respectively. In the column labeled *feature set* in Table VII, we listed feature categories in order of priority (i.e., highest F_1-score). The *compile time* was the most accurate feature category in most of the families. Each of the following behavioral categories slightly improved the score compared to using only the primary category. Using other behavioral features, we could accurately cluster samples in a family in which the compile time was manipulated, such as *Lydra, Cakl, or DarkMoon*. In particular, only behavioral features were selected for the *Lydra* family. As shown in Table VIII, only four families whose most accurate category was the compile time appeared in the reliable dataset. The number of families that used only behavioral features was seven. Our experiments on the reliable dataset indicate that we can best understand the behavior of a family and cluster samples in a family based on behavior of versions.

Fig. 4 presents the F_1-score of intrafamily clustering according to various feature sets. Individual static, dynamic, and hybrid features should not be derived, and so they include no dashed lines. Average F_1-scores of clustering using each of three features were also very low: approximately 30% for the VXHeavens dataset and approximately 70% for the reliable dataset. The performances of the static and hybrid feature appear almost the

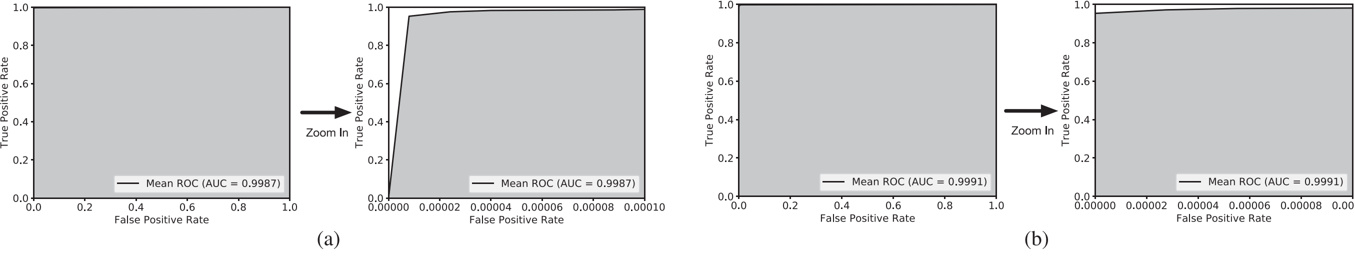

Fig. 2. (Left) ROC curve and AUC of family classification on each dataset. (Right) Zoomed-in view. (a) VXHeavens dataset. (b) Reliable dataset.

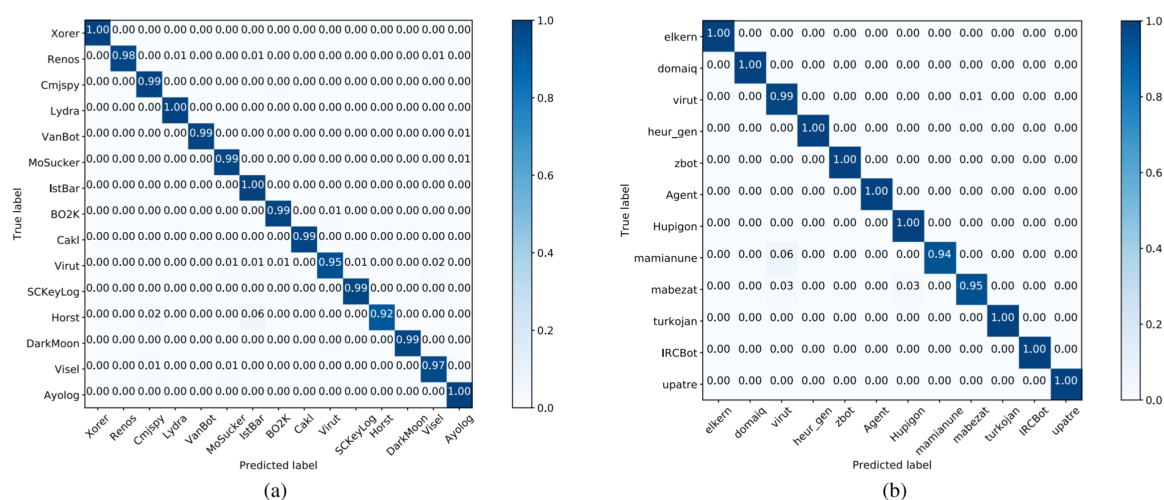

Fig. 3. Confusion matrix of family classification result on each dataset. (a) VXHeavens dataset. (b) Reliable dataset.

TABLE VII
RESULT OF FEATURE SELECTION IN THE VXHEAVENS DATASET

Family	Feature Set
Renos	compile, fileRead, fileAccess
Cmjspy	compile, serviceStarted
Lydra	registryDelete, droppedName
VanBot	compile, domainName
Cakl	compile, droppedName, command
BO2K	compile, serviceStarted
IstBar	compile, fileWrite
Virut	compile, fileDelete
SCKeyLog	compile, fileDelete
Horst	compile, command
MoSucker	compile, registryDelete
Visel	compile, registryDelete
Ayolog	compile, domainName
Xorer	compile, fileAccess, serviceStarted, serviceCreated, command, registryWrite
DarkMoon	compile, fileWrite, domainName, dnsType, fileDelete, dnsRequest, fileAccess, command, registryRead, udpSrcPort
Common	compile, fileDelete, fileWrite, serviceCreated

The row named each family denotes the family feature set and the row named common denotes the common feature set.

TABLE VIII
RESULT OF FEATURE SELECTION IN THE RELIABLE DATASET

Family	Feature Set
Elkern	api, droppedSize
Domaiq	fileAccess, registryAccess
Virut	compile, serviceStarted
Heur_gen	dnsType, domainName, mutex
Zbot	droppedSSDEEP, pcapSHA256
Agent	compile, domainIP
IRCBot	fileRead, domainName
Mamianune	compile, fileDelete
Mabezat	compile, udpTime
Turkojan	fileRead, fileDelete
Hupigon	fileDelete, domainName
Upatre	droppedSSDEEP, pcapSHA256, compile, fileAccess, fileDelete, fileRead, fileWrite, command, mutex, droppedName, registryAccess, pcapSorted
Common	droppedSSDEEP, serviceStarted

The row named each family denotes the family feature set and the row named common denotes the common feature set.

same because most values of the dynamic feature were zero or one, i.e., insignificant, but those of the static feature were very large, i.e., significant.

In the feature selection phase, the performance of the common feature set appeared comparable to family feature sets. However, the common feature set demonstrated drastic decrease in F_1-score on some families in intrafamily clustering because those families exhibit different sensitivities to the common feature set. Using common feature sets, the average F_1-score of clustering was 85.15% and 76.44% for each dataset, respectively. On the other hand, the average F_1-score of clustering based on family feature sets was 93.29% and 89.30% for each dataset, which is superior to the selected common feature set.

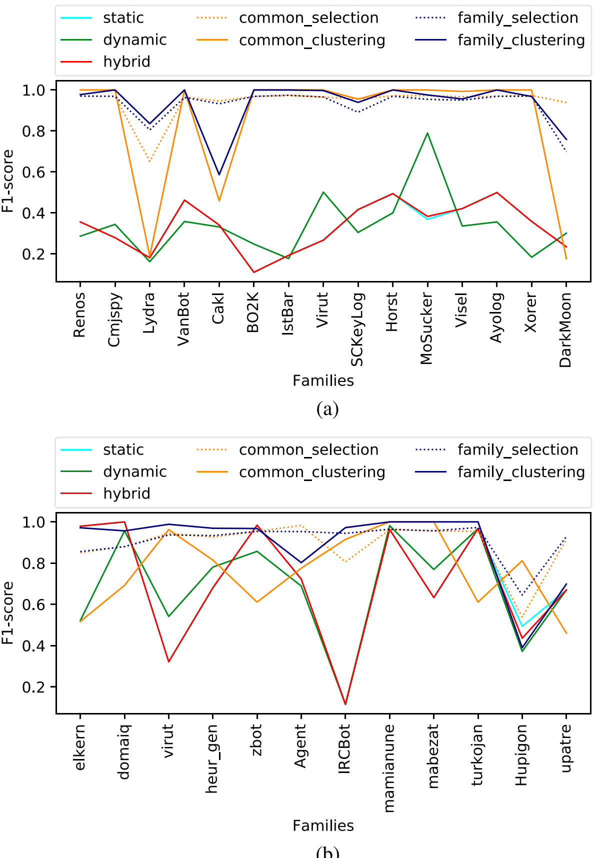

Fig. 4. F_1-score of each family in intrafamily clustering on each dataset. The dashed lines indicate the highest F_1-score of intrafamily clustering obtained while deriving feature sets, and the solid lines indicate the F_1-score of intrafamily clustering based on derived feature sets. (a) VXHeavens dataset. (b) Reliable dataset.

In the VXHeavens dataset, there were some families whose F_1-score of clustering with common feature sets outperformed the family feature set, but the difference was very trivial. The difference between two feature sets was very significant in *Lydra, DarkMoon*. The result of the reliable dataset demonstrated a similar aspect of the VXHeavens dataset. In intrafamily clustering, family feature sets outperformed the common feature set, except for the family *Hupigon*. In summary, we verified that family feature sets are more effective and more flexible for identifying new samples than the common feature set.

Compared to the prior work of Haq *et al.* [21], our clustering approach based on family feature sets seems more accurate than that in their system, which demonstrated approximately 70% of function coverage for their dataset. We tried to compare two systems in a same condition, but we could not get an open source and dataset of Haq *et al.* Although we cannot give an objective comparison, we believe that our system will also result in similar performance well on other dataset.

2) Cluster Head Selection: In this experiment, we measure the degree to which unpacked malware are accurately detected in each cluster based on cluster head selection. Table IX presents the accuracy of unpacked malware detection for the two datasets using entropy, sum of raw data size of known section, or the product of both. The results indicate that the accuracy of identifying a sample with the highest sum of raw data size of known sections is higher than that of identifying a sample with the lowest entropy or using the product of both. In addition, our method could find unpacked samples in most of the clusters.

TABLE IX
UNPACKED MALWARE DETECTION ACCURACY OF EACH FEATURE AND DATASET (%)

Dataset	**(1)** **Entropy**	**(2)** **Sum of raw data size**	**(3)** **(1) * (2)**
VX Heavens	76.04	***99.31***	95.14
Reliable	74.79	***94.12***	86.55

3) Dimensionality Reduction: In this experiment, we analyzed changes in the performance of intrafamily clustering because of dimensionality reduction, i.e., feature normalization of the behavioral features. We compared the results of normalization for behavioral features in terms of family feature sets, clustering accuracy, and memory requirements against the results that do not involve normalization.

Changes in family feature sets because of feature normalization were insignificant in both the VXHeavens and reliable datasets. For the VXHeavens dataset, seven out of 15 families changed their selected feature sets. They added, removed, and replaced one or two feature categories. Exceptionally, *DarkMoon* added six categories through normalization. Compile time retained its original priority for all families in the dataset. For the reliable dataset, four out of 12 families changed their selected feature sets. Three families replaced only one category, but *Upatre* added ten categories through normalization.

Overall, feature normalization reduced peak memory consumption in terms of both feature selection and clustering. For the VXHeavens dataset, normalization reduced peak memory consumption from 983.49 to 628.84 MiB in the feature selection phase and from 404.86 to 375.72 MiB in the clustering phase. For the reliable dataset, normalization reduced peak memory consumption from 975.50 to 444.66 MiB in the feature selection phase and from 332.70 to 333.71 MiB in the clustering phase. For both datasets, the reduced memory consumption in the feature selection phase was much larger than that in the clustering phase. Therefore, we determined that feature normalization is effective in feature selection, which tests a variety of feature category sets.

In terms of accuracy, normalization exhibits some positive effect on the result of intrafamily clustering. For the VXHeavens dataset, normalization improved the F_1-score by approximately 0.13% in the feature selection phase and 0.78% in the clustering phase. For the reliable dataset, the F_1-score was improved by approximately 0.72% in the feature selection phase and 3.97% in the clustering phase. Unlike memory consumption, normalization played a bigger role in improving the accuracy of clustering compared to feature selection. These results indicate that our method can not only significantly reduce memory consumption, but also improve clustering accuracy.

VII. IMPLICATIONS AND DISCUSSION

A. Practical Impact

In the current malware environment that mostly consists of packed malware samples, our approach plays a crucial role in version identification associated with large-scale lineage inference. We can handle packed malware based on hybrid analysis.

Moreover, we can analyze not only malware that is packed once, but also multipacked malware. In particular, three *Agent* samples and four *Elkern* samples were already packed by *UPX* before we collected the reliable dataset. Two *Agent* samples were already packed by *ASPack*. We repacked them using the other five packers by considering new variants. In terms of family classification, the F_1-scores of these families created by Agent samples and Elkern samples were 99.49% and 100%, respectively. In intrafamily clustering, the accuracy of clustering was 100% for those samples.

Our intrafamily clustering method is expected to dramatically reduce the complexity of lineage inference. Most prior lineage inference studies considered all samples to be individual versions, including packed samples. We do not have to infer the lineage of packed samples for each version; their lineage can be inferred by their unpacked samples. Although the amount of the complexity improvement of lineage inference is different, we expect that the improvement is observable in all families of our datasets. The expected computation of lineage inference using our approach for the VXHeavens dataset is at least 986 times (for the *Lydra* family) faster than existing approaches, and it is 1271 times faster for the *Horst* family. Even though the performance improvement in *Elkern* is relatively low compared to other families in the reliable dataset, we can still infer the lineage of the family 150 times faster. Lineage inference for the *Zbot* family is expected to be 1891 times faster, which denotes the largest performance improvement.

We determined that feature selection can considerably improve the accuracy of intrafamily clustering for version identification. The selection of a common feature set improved the accuracy of intrafamily clustering compared to single features such as static, dynamic, and hybrid features. The common feature set also demonstrated performance that is similar to family feature sets in the feature selection phase, which was performed on known samples. However, the common feature set could not accommodate any unknown samples, which indicates a drastic fall in F_1-scores. On the other hand, our family feature sets exhibit a clustering F_1-score of approximately 10% higher than the common feature set. Our experimental results verified that most malware families exhibit their own sensitivity of behavior, and our family feature sets represent these properties well. Although it is clear that a common feature set is required for family classification, the family feature set corresponding to a certain family is considered crucial for intrafamily analysis.

B. Limitations and Future Work

Our approach addresses some problems in lineage inference but not the lineage inference itself. As lineage inference involves identifying the sequence of malware samples, our study must be extended in this direction. However, our current concept involves identifying a version of packed malware rather than drawing timelines of malware development. Nevertheless, the future direction is very simple: clusters we created can replace elements used to construct a lineage graph that previous studies developed. It is easy to infer the lineage of clusters because we only have to infer the lineage of selected samples for each cluster.

Although we proposed a method for selecting a cluster head, there may be no unpacked samples in a cluster. Consequently, a packed sample in this cluster will be selected, and this can present several limitations involving lineage inference. Moreover, our methods are based on heuristics, and therefore, some samples may not fit into our rule. However, there are several possible solutions; for example, we can unpack a sample whose packer is obviously known by the specified packer. Even when a sample's packer is unknown, we can use a generic unpacking algorithm [10], [19], [23]. The generic unpacking algorithm cannot be successfully applied to all samples as yet, but the approach is continuously being developed. Unpacking all samples is inefficient, but unpacking one sample for each cluster only requires little complexity.

Our family classification model may not identify malware variants well when the sample's feature has been modified to a certain extent. Even in the case of new malware families, our model cannot identify them because they have not been trained. This drawback is referred to as *concept drift*, and several prior studies attempted to address this problem [12], [27], [32], [41]. In this context, a more sophisticated study should be conducted, in which new malware variants and new malware families continuously emerge in the real world.

VIII. Related Work

A. Malware Lineage Inference and Version Identification

Although our main objective of this study is solving the problem of the existing lineage inference approaches, they should be leveraged with our system together. Some malware lineage researchers focused on the premise that malware authors continue developing their work [13], [16], [21], [26], [28], [31].

Lindorfer *et al.* analyzed the lineage of self-evolving malware [31]. They used control flow graphs and used *Anubis Sandbox* as a simple generic unpacker for handling packed malware. They also correlated control flow graphs and dynamic API to compensate for each feature; however, their method could be defeated by multilayer packing. Meanwhile, our feature processing method facilitates the analysis of repacked malware.

Jang *et al.* experimented on lineage inference of malware development in the *Cyber Genome Program* [26]. They used agglomerative clustering to group similar samples and represented the lineage of each cluster using two types of graphs. They compared the performance of lineage inference using various features such as section size, binary code, and dynamic instruction.

Graziano *et al.* identified the lineage of malware submitted to Anubis Sandbox [16]. They first filtered packed samples and then used agglomerative clustering and *ssdeep* to group similar binaries. They determined whether a cluster involves malware development using hybrid features. They also used feature selection algorithms such as chi-square, gain ratio, and belief-F to enhance performance.

The main limitation of the prior three studies is the lack of version identification. Haq *et al.* considered malware version identification and extended it to lineage inference [21]. They mainly utilized function information of a binary, which can be extracted through unpacking and disassembly. They demonstrated the accuracy of version identification on benign software, but the accuracy was not high enough. Nevertheless, their concept of matching packed samples to unpacked ones inspired our approach and reduced the computational complexity of lineage inference.

B. Machine-Learning-Based Malware Analysis

Grouping malware samples is an essential step to be performed before lineage inference. Since quite long time ago,

previous studies have shown the effectiveness and preciseness of machine learning algorithms in malware analysis [2], [29], [38]. Similar to our family classification, there has also been studied about identifying the family of a detected malware sample by using clustering algorithms [5], [6], [23], [25]. Especially, Bailey *et al.* [5] and Bayer *et al.* [6] ran a sample in a virtual environment to handle polymorphic malware. Hu *et al.* also handled packed malware using only static features by applying a *generic unpacking algorithm* that runs a binary in part [23]. Those works are closely related to our approach, considering that we also leverage a clustering algorithm in intrafamily clustering.

Recently, many researchers have been trying to detect android malware by extending the existing malware analysis methods to smartphone [3], [8], [37], [42]. Meanwhile, Grosse *et al.* proposed the direction to more secure detection model, showing that ML-based malware detectors are potentially vulnerable to adversarial examples [17], [18]. This area is still growing continuously, so our concept can be not only improved, but also migrated to other platforms.

C. Feature Engineering and Feature Selection

Which malware feature should be extracted and leveraged is closely related to the performance of malware analysis. There have been various works that studied about malware features to improve the effectiveness and preciseness of malware analysis. Increased dimensionality of features can cause problems associated with computation time, and so some studies applied *feature hashing* [23], [25]. As a consequence of dimensionality reduction, those studies could considerably reduce runtime while also preserving accuracy. However, feature hashing reduces the dimensionality of the set size, which may result in collisions.

Feature selection is a good solution for simplifying feature sets and improving accuracy. Several studies applied feature selection algorithms in this regard [1], [20], [40], [43]. Information gain, chi-square, fisher score, and semantic network can be used to select well-performing feature candidates. In particular, Ahmadi *et al.* [1] derived a feature set by using the *forward stepwise selection algorithm*. We applied the same algorithm, but we derived family feature sets instead of the common feature set that Ahmadi *et al.* derived.

IX. Conclusion

As the amount of packed malware grows with each passing year, lineage inference is an inevitable significant challenge that must be considered. In this article, we proposed a novel approach that executes stepwise feature processing and version identification. The various methods of feature processing introduced in this article assisted in mitigating the problem of malware packing. We were able to not only classify malware into families, but also reduce the computational complexity of feature selection using the forward stepwise selection algorithm. In addition, we improved the accuracy of intrafamily clustering by creating family feature sets. Our experiments on two datasets verified the reliability of framework evaluation, and current experiments demonstrate that we can effectively handle a significant amount of packed malware for lineage inference. We also discussed the limitations of our approach and possible directions for future work. Finally, the proposed method mitigates the feature and packing problems in terms of accuracy and calculation complexity that prior lineage inference studies experienced. Therefore, it would be a promising direction to adopt our approach to the traditional lineage inference.

References

[1] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto, "Novel feature extraction, selection and fusion for effective malware family classification," in *Proc. 6th ACM Conf. Data Appl. Secur. Privacy*, 2016, pp. 183–194.

[2] B. Anderson, C. Storlie, and T. Lane, "Improving malware classification: Bridging the static/dynamic gap," in *Proc. 5th ACM Workshop Secur. Artif. Intell.*, 2012, pp. 3–14.

[3] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "DREBIN: Effective and explainable detection of android malware in your pocket," in *Proc. NDSS Symp.*, 2014, pp. 23–26.

[4] AVTEST, "Malware statistics & trends report." [Online]. Available: https://www.av-test.org/en/statistics/malware/, Accessed on: Dec. 29, 2019.

[5] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario, "Automated classification and analysis of Internet malware," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*, 2007, pp. 178–197.

[6] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2009, pp. 8–11.

[7] BradSpengler and 102 contributors, "Cuckoo-modified, modified edition of cuckoo." [Online]. Available: https://github.com/spender-sandbox/cuckoo-modified, Accessed on: Dec. 29, 2019.

[8] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for android," in *Proc. 1st ACM Workshop Secur. Privacy Smartphones Mobile Devices*, 2011, pp. 15–26.

[9] E. Carrera and G. Erdélyi, "Digital genome mapping–advanced binary malware analysis," in *Proc. Virus Bull. Conf.*, 2004, vol. 11, pp. 187–197.

[10] B. Cheng *et al.*, "Towards paving the way for large-scale windows malware analysis: Generic binary unpacking with orders-of-magnitude performance boost," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 395–411.

[11] Cuckoo Sandbox, "Automated malware analysis." [Online]. Available: https://cuckoosandbox.org/, Accessed on: Dec. 29, 2019.

[12] A. Deo, S. K. Dash, G. Suarez-Tangil, V. Vovk, and L. Cavallaro, "Prescience: Probabilistic guidance on the retraining conundrum for malware detection," in *Proc. ACM Workshop Artif. Intell. Secur.*, 2016, pp. 71–82.

[13] T. Dumitras and I. Neamtiu, "Experimental challenges in cyber security: A story of provenance and lineage for malware," in *Proc. USENIX Workshop Cyber Secur. Exp. Test*, 2011, vol. 11, pp. 2011–2019.

[14] A. Feizollah, N. B. Anuar, R. Salleh, and A. W. A. Wahab, "A review on feature selection in mobile malware detection," *Digit. Investigation*, vol. 13, pp. 22–37, 2015.

[15] G DATA Security Blog, "In 2017 every 4.2 seconds a new malware specimen emerges." [Online]. Available: https://www.gdatasoftware.com/blog/2017/04/29666-malware-trends-2017, Accessed on: Dec. 29, 2019.

[16] M. Graziano, D. Canali, L. Bilge, A. Lanzi, and D. Balzarotti, "Needles in a haystack: Mining information from public dynamic analysis sandboxes for malware intelligence." in *Proc. USENIX Secur. Symp.*, 2015, pp. 1057–1072.

[17] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial perturbations against deep neural networks for malware classification," 2016, *arXiv:1606.04435*.

[18] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial examples for malware detection," in *Proc. Eur. Symp. Res. Comput. Secur.*. 2017, pp. 62–79.

[19] F. Guo, P. Ferrie, and T.-C. Chiueh, "A study of the packer problem and its solutions," in *Proc. 11th Int. Symp. Recent Adv. Intrusion Detection*, 2008, pp. 98–115.

[20] H.-S. Ham and M.-J. Choi, "Analysis of android malware detection performance using machine learning classifiers," in *Proc. Int. Conf. ICT Convergence*, 2013, pp. 490–495.

[21] I. Haq, S. Chica, J. Caballero, and S. Jha, "Malware lineage in the wild," *Comput. Secur.*, vol. 78, pp. 347–363, 2018.

[22] V. Heaven, 2016. [Online]. Available: http://83.133.184.251/virensimulation.org/

[23] X. Hu, K. G. Shin, S. Bhatkar, and K. Griffin, "Mutantx-s: Scalable malware clustering based on static features," in *Proc. USENIX Annu. Tech. Conf.*, 2013, pp. 187–198.

[24] G. Jacob, P. M. Comparetti, M. Neugschwandtner, C. Kruegel, and G. Vigna, "A static, packer-agnostic filter to detect similar malware samples," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2012, pp. 102–122.
[25] J. Jang, D. Brumley, and S. Venkataraman, "Bitshred: Feature hashing malware for scalable triage and semantic analysis," in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, 2011, pp. 309–320.
[26] J. Jang, M. Woo, and D. Brumley, "Towards automatic software lineage inference," in *Proc. USENIX Secur. Symp.*, 2013, pp. 81–96.
[27] R. Jordaney *et al.*, "Transcend: Detecting concept drift in malware classification models," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 625–642.
[28] M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida, "Malware phylogeny generation using permutations of code," *J. Comput. Virology*, vol. 1, no. 1, pp. 13–23, 2005.
[29] J. Z. Kolter and M. A. Maloof, "Learning to detect and classify malicious executables in the wild," *J. Mach. Learn. Res.*, vol. 7, no. Dec, pp. 2721–2744, 2006.
[30] Y. Li *et al.*, "Experimental study of fuzzy hashing in malware clustering analysis," in *Proc. 8th USENIX Workshop Cyber Secur. Exp. Test*, 2015, p. 52.
[31] M. Lindorfer, A. Di Federico, F. Maggi, P. M. Comparetti, and S. Zanero, "Lines of malicious code: Insights into the malicious software industry," in *Proc. 28th Annu. Comput. Secur. Appl. Conf.*, 2012, pp. 349–358.
[32] M. Lindorfer, M. Neugschwandtner, and C. Platzer, "MARVIN: Efficient and comprehensive mobile app classification through static and dynamic analysis," in *Proc. 39th IEEE Annu. Comput. Softw. Appl. Conf.*, 2015, vol. 2, pp. 422–433.
[33] R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," *IEEE Secur. Privacy*, vol. 5, no. 2, pp. 40–45, Mar./Apr. 2007.
[34] J. Ma, J. Dunagan, H. J. Wang, S. Savage, and G. M. Voelker, "Finding diversity in remote code injection exploits," in *Proc. 6th ACM SIGCOMM Conf. Internet Meas.*, 2006, pp. 53–64.
[35] Python Software Foundation, "Welcome to python.org." [Online]. Available: https://www.python.org/, Accessed on: Dec. 29, 2019.
[36] B. Ruttenberg *et al.*, "Identifying shared software components to support malware forensics," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2014, pp. 21–40.
[37] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "Madam: Effective and efficient behavior-based android malware detection and prevention," *IEEE Trans. Dependable Secure Comput.*, vol. 15, no. 1, pp. 83–97, Jan./Feb. 2018.
[38] M. G. Schultz, E. Eskin, F. Zadok, and S. J. Stolfo, "Data mining methods for detection of new malicious executables," in *Proc. IEEE Symp. Secur. Privacy*, 2000, pp. 38–49.
[39] scikit-learn, "Machine learning in python." [Online]. Available: http://scikit-learn.org/stable/index.html, Accessed on: Dec. 29, 2019.
[40] A. Shabtai and Y. Elovici, "Applying behavioral detection on android-based devices," in *Proc. Int. Conf. Mobile Wireless Middleware, Oper. Syst. Appl.*, 2010, pp. 235–249.
[41] A. Singh, A. Walenstein, and A. Lakhotia, "Tracking concept drift in malware families," in *Proc. 5th ACM Workshop Secur. Artif. Intell.*, 2012, pp. 81–92.
[42] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual API dependency graphs," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 1105–1116.
[43] Z. Zhu and T. Dumitraş, "FeatureSmith: Automatically engineering features for malware detection by mining the security literature," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 767–778.

Leo Hyun Park received the B.S. degree in computer engineering from Kwangwoon University, Seoul, South Korea, in 2017. He is currently working toward the Ph.D. degree with Information Security Laboratory, Yonsei University, Seoul.

His research interests include information security and privacy, malware analysis, usable security, artificial intelligence security, machine learning algorithms, and adversarial machine learning.

Jungbeen Yu received the B.S. degree in information security from Daegu Catholic University, Gyeongsan, South Korea, in 2016, and the M.S. degree in information security from Yonsei University, Seoul, South Korea, in 2018.

His research interests include information security and privacy, malware analysis, artificial intelligence security, and machine learning.

Hong-Koo Kang received the B.S., M.S., and Ph.D. degrees in computer science from Konkuk University, Seoul, South Korea, in 2002, 2004, and 2009, respectively.

From 2009 to 2010, he was a Lecture Professor of Computer Science with Konkuk University, Seoul. He is currently a General Researcher of Security Threat Response R&D with Korea Internet and Security Agency, Naju, South Korea. His research interests include malware analysis, web security, cyber threat intelligence, and artificial intelligence security.

Taejin Lee received the B.S. degree from the Pohang University of Science and Technology, Pohang, South Korea, in 2003, the M.S. degree from Yonsei University, Seoul, South Korea, in 2008, and the Ph.D. degree from Ajou University, Suwon, South Korea, in 2017.

He has conducted R&D research with Korea Internet and Security Agency, Naju, South Korea, from 2003 to 2016. He is currently a Professor with Hoseo University, Asan, South Korea. His research interests include malware analysis, network security, and artificial intelligence security.

Taekyoung Kwon (M'02) received the B.S, M.S., and Ph.D. degrees in computer science from Yonsei University, Seoul, South Korea, in 1992, 1995, and 1999, respectively.

He is currently a Professor of Information Security with Yonsei University, Seoul, where he is a Director of the Information Security Lab. From 1999 to 2000, he was a Postdoctoral Research Fellow with the University of California, Berkeley, CA, USA. From 2001 to 2013, he was a Professor of Computer Engineering with Sejong University, Seoul. His research interests include authentication, cryptographic protocols, network security, software and system security, usable security, artificial intelligence security, and adversarial machine learning.

Dr. Kwon is on the Director Board of the Korea Institute of Information Security and Cryptology and on the Editorial Committee of the Korean Institute of Information Scientists and Engineers. He is a member of the Association for Computing Machinery and USENIX.