Detecting and Classifying Self-Deleting Windows Malware Using Prefetch Files

Adam Duby
United States Military Academy
adam.duby@westpoint.edu

Teryl Taylor
IBM Research
terylt@ibm.com

Gedare Bloom, Yanyan Zhuang
University of Colorado Colorado Springs
{gbloom, yzhuang}@uccs.edu

2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC) | 978-1-6654-8303-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/CCWC54503.2022.9720874

***Abstract*—Malware detection and analysis can be a burdensome task for incident responders. As such, research has turned to machine learning to automate malware detection and malware family classification. Existing work extracts and engineers static and dynamic features from the malware sample to train classifiers. Despite promising results, such techniques assume that the analyst has access to the malware executable file. Self-deleting malware invalidates this assumption and requires analysts to find forensic evidence of malware execution for further analysis. In this paper, we present and evaluate an approach to detecting malware that executed on a Windows target and further classify the malware into its associated family to provide semantic insight. Specifically, we engineer features from the Windows prefetch file, a filesystem forensic artifact that archives process information. Results show that it is possible to detect the malicious artifact with 99% accuracy; furthermore, classifying the malware into a fine-grained family has comparable performance to techniques that require access to the original executable. We also provide a thorough security discussion of the proposed approach against adversarial diversity.**

***Index Terms*—Malware analysis, forensics, malware classification, prefetch**

I. Introduction

Malware analysis is an integral aspect of incident response. Security analysts need to quickly identify the extent of damage done by malware to inform mitigation and prioritize defensive efforts. Two important initial steps in this identification process are (1) to detect evidence of malware execution, and (2) to classify detected malware into descriptive families. Rapid and accurate detection and classification are vital to limiting the damage and recovering from an attack.

The tools used by analysts to automatically detect and classify malware are based largely on machine learning techniques. To detect malware, the machine learning algorithms will learn models of benign software and malware. The selection and availability of features are vital to the effective use of these automation tools. Such features include information extracted from *static* binary executable files [38], [39] and from the *dynamic behavior* of malware/software execution [27].

Unfortunately, most existing research in malware classification assumes that the malware sample is available for analysis. This is often not the case. For example, advanced persistent threat (APT) groups are known to conduct targeted campaigns and then self-delete the deployed malware (along with the usual activity of cleaning up log files, etc.)[1]. Self-deletion requires analysts to identify *residual artifacts* left behind by the malicious activity [13], such as network logs on uninfected machines and routers [34] or memory and filesystem forensic data left behind on a host [9], [20]. Identification and evaluation of the efficacy of such forensic feature sets in malware detection and family classification is an open problem [37].

In this paper, we describe an approach to detect evidence of malware execution post attack, and then perform malware family classification on the suspected malware without assuming access to the original malware sample. The value of malware family classification is that it informs analysts on the behavior of the malware. As such, knowing the family improves the effectiveness of incident response.

We utilize Windows *prefetch files* to engineer features that are resilient to malware self-deletion. The Windows operating system (OS) creates prefetch files when it loads and executes a program in order to cache the locations of the program's referenced files, such as its dynamically linked libraries (DLLs), so that future program loading is faster. Inspired by use of prefetch files as residual artifacts in digital forensics [9], we extract a *prefetch feature set* for each process by comprising the set of references from the process's respective prefetch file to detect and classify malware. Prefetch files are available by default on Windows systems. Thus, our approach requires minimal effort for analysts to use. We engineer the features to reduce susceptibility to overfitting and concept drift (i.e., malware evolution), and train a k-nearest-neighbor (kNN) classifier to detect if a prefetch file is evidence of malware execution. Then we apply a custom classifier based on Jaccard similarity between sets to classify the suspected malware into its closest family.

Classifying malware into a known family exposes the malware's underlying behavior because malware within the

This work is supported in part by NSF grants OAC-2115134, OAC-1920462, OAC-2001789, CNS-2046705, and Colorado State Bill 18-086. The views expressed in this paper are those of the authors and do not reflect the official policy or position of the United States Military Academy, the United States Army, the Department of Defense, or the United States Government.

[1]https://attack.mitre.org/groups/G0016/

same family share a common subset of semantics. For example, classifying a sample as WannaCry ransomware informs analysts that the sample exhibits behavior representative of WannaCry, e.g., file encryption and propagation via Server Message Block vulnerabilities [7]. If a common feature is removed from this subset, the malware's semantics are degraded because it fundamentally changes the behavior of the malware [29]. This is akin to a monotonic increase constraint, where the feature set of a variant may increase (i.e., features added), but the shared features common to the family cannot be removed due to loss of functionality [15]. We define the enforcement of a minimum feature set in malware classification as a *semantic preservation constraint*, and enforce this constraint in our family classifier.

We evaluate our approach for malware detection and family classification using prefetch features with a dataset containing 48 malware families and over 4,200 benign files captured from live production systems. Our detector accurately detected evidence of malware execution with over 99% accuracy, and the family classifier achieved results competitive with other techniques that require full access to the malware and extraction of more features, such as application programming interface (API) calls. Finally, we show that the Jaccard similarity-based family classifier with semantic preservation constraints provides better performance over an ensemble machine learning classifier.

This paper makes the following contributions:

- We describe an approach to detect evidence of malware execution despite malware self-deletion, and classify the malware into its most probable family;
- We describe an approach to enforce semantic preservation constraints in the feature set;
- We evaluate our approach and demonstrate improved performance over state-of-the-art techniques.

The rest of the paper is organized as follows. Section II explains background knowledge and Section III reviews the related work. We describe our approach in Section IV and present the evaluation with experimental results in Section V. In Section VI we discuss the benefits and limitations of our approach, and we conclude with Section VII.

II. Background

Malware Self-Deletion. Malware can delete itself after execution to evade detection and thwart analysis. The MITRE ATT&CK framework defines such behavior as *Indicator Removal on Host::File Deletion* (T1070.004)[2]. We queried the ATT&CK repository and found that 29.8% of threat actors and 33.7% of malware campaigns have utilized self-deletion tactics. As such, our research focuses on malware analysis to inform incident responders who lack access to the original executable file.

Prefetch Files. On Windows, when a process loads and for the first ten seconds that it executes, the OS monitors file references that are made (e.g., loaded DLLs, subprocess execution, language/font files) and maps the full path of those references into the application's respective prefetch file. Each executed process has its own separate prefetch file, and each machine can store up to 1024 prefetch files for recent applications. Subsequent launches of the same application will load (i.e., prefetch) dependencies from the filesystem into memory with the process.

Threat Model. Our threat model considers an evasive self-deleting malware. This constrains analysts to rely exclusively on digital forensic techniques to (1) hunt for evidence of malware execution (i.e., malware detection), and (2) classify the malware into a descriptive family to inform response efforts.

III. Related Work

Malware classification techniques can be loosely categorized into two approaches: static and dynamic. A static approach extracts features from the raw malware file (i.e., opcodes). By contrast, a dynamic approach extracts features during program execution (i.e, API calls). Both approaches assume the analyst has access to the malware binary.

Static Malware Analysis. Approximate matching [5] techniques, or fuzzy hashing, have been used to estimate similarity between malware by piece-wise hashing their executable files and comparing the overlap, like ssdeep [18], sdhash [31], and tlsh [25]. Unfortunately, fuzzy hashing is highly sensitive to minor deviations in program syntax [12], [26], [32]. Malware can obfuscate itself, creating variants that appear wildly different on the surface while retaining semantic similarity, rendering fuzzy hashing techniques weak for classification.

Some static techniques extract certain features instead of processing the entire file. This can include structural file information [33], [40], information from the file header [39], or information obtained directly from opcodes [14], [36], [38]. Common approaches leverage dependency-based information (i.e., libraries) [8], [24]. Similarly, ImpHash is used to create a digest of a program's import address table, which can be used to find samples that import the same libraries [22]. Static features are generally easy to extract from the raw file. However, static features are severely limited when faced with packed (i.e, heavily obfuscated) malware [1].

Dynamic Malware Analysis. Executing suspected malware samples allows collection of dynamic features at runtime such as API and system calls [2], [6], [27], call graphs [30], and resource usage [16], [23], [35]. Identifying and extracting features by running the malware however can be a complex and resource-intensive task. Dynamic analysis also does not guarantee complete code coverage, as some malware may terminate prematurely, thwarting the extraction of a complete feature set. As with static approaches, dynamic analysis still requires access to (complete) malware samples.

Digital Forensics Techniques. One way to overcome the limitation of requiring access to the malware is to use forensic techniques, which work without an executable program. For example, memory forensics artifacts extracted from process memory can be used to cluster similar malware [19], [20].

[2]https://attack.mitre.org/techniques/T1070/004/

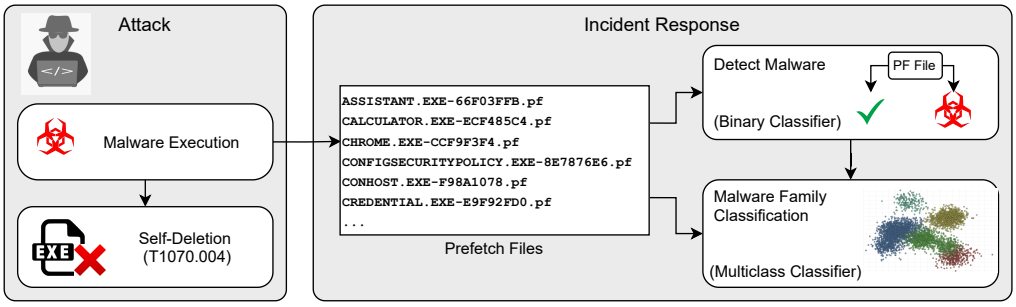

Fig. 1. Approach overview.

Process memory provides an abundance of information, but memory extraction is costly in time and space, and it requires specialized tools and expertise. Forensics also degrade with time. For example, the OS reclaims and reuses memory after process termination so memory extraction techniques work best while the process to analyze is still running. Complementary to compromised hosts, network forensics techniques using log and packet analysis [34] or traffic pattern analysis can detect malware [21]. However, these techniques require excessive storage (to collect full network traces) and may still miss malware that have no obvious network presence.

Prefetch Techniques. Alsulami et al. [4] trained a logistic regression model using features extracted from prefetch files to detect malware (i.e., binary classification). Prefetch files have also been used to train a convolutional recurrant neural network to classify malware into five course-grained type labels: adware, backdoor, trojan, virus, and worm [3]. However, such type labels are too coarse-grained to provide actionable threat intelligence for incident response.

In this work, we evaluate our approach under a more challenging, yet usable, label space of 48 malware families that represent a breadth of malware functionality. Analysts can infer malware behaviors and prioritize response efforts from family labels. Further, we suggest that existing prefetch techniques are prone to overfitting, which may lead to unrealistic performance. As such, we implement a feature selection pipeline that reduces the features' disposition to overfitting and susceptibility to changes over time (e.g, concept drift). Finally, existing techniques do not guarantee semantic preservation constraints as desired for malware family classification [10]. Our approach leverages a set-theoretic technique to enforce such techniques to reduce mislabeling of instances that may teeter along a classifier's decision boundary.

IV. Approach

Our approach extracts a novel set of features from the prefetch files. The technique involves three phases as shown in Figure 1: (1) feature extraction, selection, and engineering; (2) malware detection; and (3) malware family classification.

A. Feature Extraction, Selection, and Engineering

After a program is executed, we extract its loaded libraries from its prefetch file, which form this program's *prefetch feature set*. We apply a selection and normalization process to remove brittle features based on the following criteria: vulnerability to concept drift, disposition to overfitting, low variance and high correlation with other features.

Reducing concept drift. Concept drift is when features in datasets decay over time due to underlying changes in the malware [17]. Such drift is observed in loaded libraries, and consequently in the raw prefetch feature sets. For example, a program's bitness (i.e., 32-bit versus 64-bit) can change over time. Bitness is exposed through the presence of Windows-on-Windows (WoW) subsystem DLLs (i.e., `wow64.dll`). Since this is a consequence of computer architecture and not representative of the program's semantics, WoW DLLs are excluded. We also normalize DLLs that provide a version control numbering convention in their names.

Overfitting avoidance. Overfitting can occur when a dataset contains features that leak the target class at training time [41] or when features can be uniquely associated with the target class. We remove such sources of leakage and overfitting from our feature set. For example, we do not consider executable names as part of our analysis even though they are present in prefetch files. This is because malware authors can easily change the names to arbitrary strings. Furthermore, malware families can also drop custom DLLs and EXEs, which can also lead to overfitting. Finally, we ignore features in both the malware and benign datasets that make use of locally (or globally) unique identifiers (LUID/GUID). These values can change between machines, and models may overvalue the importance of such features if the training data was generated from instances running on the same machine.

Identifying features of little information. Features that are present in almost every prefetch file were removed since they do not aid classifiers in discerning between labels. Examples include `ntdll.dll` and `kernel32.dll`. Similarly, we remove redundant features that have a pairwise Pearson correlation coefficient greater than 0.95. When a process loads a DLL, the OS loader recursively walks the dependency chain to load the DLL's dependencies. Consequently, DLLs with dependencies on other DLLs create clusters of highly correlated features (e.g., `zlib.dll` and `zlibwapi.dll`).

Dimensionality reduction. The feature selection and engi-

neering pipeline presented above reduced feature dimensionality from 4,321 unique features (i.e., references) down to 1,381 across our entire dataset. The final per-process feature set is represented as an unordered set, or a bag-of-words (BoW), of the references made by a process and is used to train our malware detector and family classifier.

B. Malware Detection

The malware detector identifies which process' prefetch file contains evidence of malicious behavior. To do so, we train a k-nearest-neighbor (kNN) model using the Jaccard distance metric and $k = 5$. Each machine stores up to 1,024 prefetch files that are each associated with the system's most recent 1,024 processes. To distinguish which prefetch file(s) are indicative of malware, the processed features are extracted from each prefetch file and they are clustered against our labeled dataset of malware and benign software. The 5 closest software instances dictate the file's membership (malware or benign) based on a majority vote. Indeed, prefetch files themselves are not malware, but our approach detects the execution of a malware process based on the forensic evidence left by each process's prefetch file.

C. Malware Family Classification

Now that the malicious process's prefetch file has been identified, we use its prefetch feature set to classify the malware into a known malware family. The features learned from the malware dataset are used to compute a similarity metric that extends Jaccard similarity (JS) to guarantee semantic preservation, described as follows.

We ensure semantic preservation by finding the *minimum feature set* for each family, which is the intersection of all prefetch feature sets of malware in the same family. This minimum feature set follows the intuition that each malware variant within a family share a common subset of behaviors. By guaranteeing the minimum feature set is met before computing similarity, we enforce the assumption that removing core features degrades program semantics.

Let Y be the set of malware families. For each malware family $y \in Y$, we find three signatures: the minimum feature set ($y_{min} = \bigcap_{i=1}^{n} y_i$, where n is the number of malware samples in y), the union of all features ($y_{max} = \bigcup_{i=1}^{n} y_i$), and the symmetric difference of the two ($y_{min} \triangle y_{max}$).

If x is the set of features from an unknown malware instance, we first verify if x is a proper superset of y_{min}. If not, we disregard y as a prospective family because it violates semantic preservation. We also check if the symmetric difference between x and y_{min} is a subset of the symmetric difference of y_{min} and y_{max}. If not, we disregard y because the feature set of x contains features not representative of the variance of family y.

This approach yields a set Z of prospective families where $|Z| < |Y|$, filtering the search space for malware family classification. From Z, the family with the highest average pairwise Jaccard similarity with the unknown malware sample is predicted to be the family. The process is described in Equation 1.

$$S(x,y) = \begin{cases} \frac{|x \cap y|}{|x \cup y|} & \text{if } (x \supset \bigcap_{i=1}^{n} y_i) \text{ and } (x \triangle \bigcap_{i=1}^{n} y_i) \subset (\bigcap_{i=1}^{n} y_i \triangle \bigcup_{i=1}^{n} y_i); \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

V. Evaluation

In this section, we first describe the dataset (Section V-A), then present our experimental design and results (Section V-B).

A. Dataset

Our dataset contains 4,442 prefetch files representing 4,442 unique malware process executions collected from 48 malware families executed in a sandbox hypervisor, and 4,296 presumed benign prefetch files collected from twenty Windows computers in a university computer lab. Benign applications collected include common Microsoft Office products, text editors, web browsers, and various integrated development environments (IDEs). The malware families capture a variety of malware behavior, including cyber espionage (Duqu 2.0), proxy-enabling click fraud (Nodersok), and self-propagating worms that install backdoors (EternalRocks). We executed each malware in a Windows 10 virtual machine and extracted their respective prefetch feature sets as described in Section IV-A.

To ensure our features are not products of their environment and will scale to a deployed environment, we examined a subset of the benign applications in the sandbox. After filtering and processing the features as described in Section IV-A, we observed no difference in features produced in production from those produced in the hypervisor.

B. Experiments and Results

We present in the following the experimental design and results for malware detection and family classification. Each test of the malware detector (Section IV-B) either produces a true positive t_p (malware is correctly classified), a true negative t_n (benign software is correctly classified), a false positive (f_p) error (misclassifies benign software as malware), or a false negative (f_n) error (misclassifies malware as benign software). Our detector results are expressed in terms of false positive rate (FPR), true positive rate (TPR), and accuracy. Precision measures the quality or exactness of the approach, and is calculated as $p = \frac{t_p}{t_p + f_p}$. Recall measures the sensitivity and completeness of the approach, and is calculated as $r = \frac{t_p}{t_p + f_n}$. The F-Score combines precision and recall by taking their harmonic mean such that $F = \frac{2 * p * r}{p + r}$. The malware classifier results are presented in terms of F-Score and compared to related classification approaches.

1) Malware Detection: For malware detection, both the malicious and benign data was leveraged. However, the distribution of classes in our dataset is not representative of the real-world, posing a risk of spatial bias [28] in results if the distribution is not corrected. Assuming there was one malware incident per machine at any given time, the expected ratio of malware to benign software is 1 to 1023 respectively, since Windows stores up to 1024 prefetch files. Therefore, we oversample the majority class (benign software) to create a distribution representative of expected real-world scenarios. Then we ran our kNN detector over the modified dataset using a stratified 10-fold cross-validation. Our results are compared with a logistic regression (LR) detector from prior work [4] that uses our feature set.

As shown in Table I, kNN provides improved false positive rates and false negative rates over LR. We also compare these models to a *base model*: after oversampling the majority class to meet the approximate 1:1023 distribution representative of the real world, the base model simply classifies all samples as benign and still achieves near perfect accuracy due to the infrequency of the malware class. However, this comes with a high false negative rate and zero recall for malware. Although our proposed model's accuracy is less than that of the base model, our goal is to detect the minority class malware, i.e., to reduce the false negative rate and increase the recall. As shown in Table I, our goal was attained.

2) Family Classification: For malware family classification, we used two different splits in the malware portion of the dataset: a random split, and a temporal split. The random split selected 75% of the data for training, and 25% for testing. The temporal split was based on compile timestamps such that only the testing set contains the more recent variants from each family. Comparing the splits allows us to evaluate our approach for temporal bias [28], where a classifier is leaked future knowledge during training. The random testing split was used to find the y_{min} and y_{max} sets for each family's semantic preservation constraints. Then we ran our family classifier from Section IV-C and compare the results with three alternative approaches: fuzzy hashing, machine learning with common dynamic features, and an ensemble machine learning classifier using our prefetch features. The results are shown in Table II.

Fuzzy Hashing. Our family classifier outperformed whole-file fuzzy hashing techniques that require access to the malware executable. Specifically, it provides a significant boost in recall. This is consistent with observations in related work that demonstrate how common fuzzy hashing techniques struggle with recalling variants where the underlying file syntax and structure are diverse [26]. The prefetch feature set is agnostic of the underlying syntax and low-level details of the malware, improving robustness in recall.

Machine Learning Techniques. We also compare our family classification approach to machine learning techniques that leverage rich dynamic features. To perform such comparisons, we extracted API calls and memory-mapped files from malware by executing the samples in a Cuckoo sandbox[3]. The features were then used to train an ensemble classifier that uses kNN and random forests. As shown in Table II, machine learning classifiers with access to these rich feature sets do provide improved performance over the prefetching approach. However, our approach is more flexible, and can be used with or without a malware sample, making it an ideal forensic tool.

TABLE I
MALWARE DETECTION RESULTS (FPR: FALSE POSITIVE RATE; FNR: FALSE NEGATIVE RATE).

Model	Accuracy %	FPR	FNR	Recall
kNN	99.92 $\pm$ 0.05	0.0005	0	0.98
LR	97.60 $\pm$ 0.31	0.02379	0.00057	0.97
Base	99.99	0	1	0

Ensemble Classifier. Finally, we evaluate the prefetch feature sets' use in an ensemble machine learning classifier that uses kNN and random forests. Our approach from Section IV-C provides a slight performance boost over an ensemble machine learning classifier. In addition to a performance hit, the machine learning classifier does not guarantee semantic preservation. Further, a machine learning approach requires additional feature preprocessing to vectorize the feature space, and additional overhead in extending the training dataset. If a new family is added to a dataset, the approach from Equation 1 only needs to calculate a few simple set operations to obtain y_{min}, y_{max}, and the symmetric difference of the two. By contrast, the machine learning classifier requires retraining the entire model, which is time consuming.

VI. SECURITY DISCUSSION

The *attack surface* of our approach was analyzed by surveying common linking methods and manually inspecting their influence on prefetch files. Our analysis provides critical insight into feature *robustness*, i.e., resilience to feature perturbation. Adversarial malware interferes with classification by perturbing the features [29]. For the prefetch feature set, such perturbation is achieved by adding or removing file references. Adversarial attacks must be realizable while preserving the desired semantics of the malware. As such, attackers cannot trivially remove file references from the malware, as it may break functionality. An adversary must exploit the actual feature extraction process, i.e., the attack must manipulate how the malware loads libraries to avoid their archival in the prefetch file. The results are summarized in Table III.

Implicit and Explicit Linking. Loaded libraries (i.e, DLLs) can be linked and loaded at load time or run time. Load time is when the OS reads an executable from disk and allocates memory. Run time is process execution. *Implicitly linked* DLLs are dynamically loaded at process load time, whereas *explicitly linked* DLLs are linked at runtime. Since the prefetch service monitors loads made by each process at

[3]https://github.com/cuckoosandbox

TABLE II
CLASSIFICATION RESULTS.

Technique	Feature	Feature Representation	Algorithm	F-Score
Fuzzy Hashing (Static)	Import Address Table (IAT)	ImpHash [22]	Pairwise Comparison	.30
	Whole File	Piecewise File Chunks	ssdeep [18]	.49
	Whole File	Byte Stream	TLSH [25]	.36
	Whole File	Piecewise File Chunks	sdhash [31]	.44
Machine Learning Classifiers (Dynamic)	API Calls	Dummy Encoded	Ensemble Classifier	.81
	API Calls	Bag of Words	Ensemble Classifier	.85
	Memory-Mapped Files	Dummy Encoded	Ensemble Classifier	.78
Our Approach	Prefetch Feature Set	Dummy Encoded	Ensemble Classifier	.80
	Prefetch Feature Set	Set	Equation 1 (Section IV)	.82

TABLE III
ATTACK SURFACE.

Linking Method	Feature Capture
Implicit Linking	✓
Explicit Linking	✓
Compiler Delay Load	✓
Explicit Delay Load $>$ 10 sec	✗
Reflective Load	✗

load time and during the first ten seconds of run time, implicitly linked DLLs and explicitly linked DLLs are captured in the prefetch file, as shown in the first two rows of Table III.

Delay and Reflective Loading. Delay loading implicitly links DLLs at run time instead of load time and can be implemented via certain compiler flags[4] (compiler delay load), or manually by the programmer (explicit delay load). Compiler delay loaded DLLs remain in a program's import address table and are therefore captured in the prefetch file. However, *explicit delay loading* that occurs after the ten second monitoring period are neglected from the prefetch file. As a result, adversary can artificially force a delay in an explicit load, e.g., using a `Sleep()` function before the explicit load. This is a limitation inherit in the design of the prefetch service. The Windows OS could make the `LoadLibrary()` record library loads to the prefetch file, mitigating this design constraint.

Attackers can also bypass the OS's native loader through reflective DLL loading [11]. The DLL is copied into memory, and a custom minimal reflective loader loads the library, bypassing the OS's normal data structures that register loaded libraries. Therefore, reflectively loaded DLLs bypass the prefetch file. However, Windows Defender alerts on reflective loading by monitoring for irregular memory mapped sections. This deters adversaries from pursuing such approaches.

Miscellaneous. An attacker could also disable the prefetching capability on the system. The Windows Superfetch service enables prefetching for applications, and it can be disabled by setting the `PrefetchParameters` key in the Memory Management registry to zero. Defenders can alert on such activity to deter this technique.

[4]Visual Studio supports delay loading via the `\DELAYLOAD` linker option.

Attackers can link and load superfluous DLLs to the malware process, but this inherits some risks to the malware's functionality. For example, the attacker must assume that the superflous DLLs reside on the target machine. A failure to load a DLL at process load time can disrupt the malware's intended capabilities. As discussed earlier, it is not trivial for adversaries to perturb the feature set through subtraction. Instead, adding noise is a more realizable adversarial approach. We leave this research for future work.

Although we identified several attack vectors for adversaries to perturb the features, we did not observe such activity pervasively in our dataset. Specifically, we observed that 1.8% of instances have DLLs in process memory that are not resident in the prefetch file. Therefore, the prefetch feature set provides a reasonable approximation of a process's dependencies.

VII. CONCLUSIONS

This work introduces an approach for improving Windows incident response and triage of malware-compromised machines based on prefetch files that can be recovered forensically even after the malware itself is deleted and unavailable. Our approach leverages library dependencies generated by the OS loader as features to build both a malware detector and family classifier. Experimental results show that using these features derived without direct access to malware samples can achieve comparable classifier performance as state-of-the-art machine learning classifiers that do require the malware sample.

REFERENCES

[1] H. Aghakhani, F. Gritti, F. Mecca, M. Lindorfer, S. Ortolani, D. Balzarotti, G. Vigna, and C. Kruegel. When malware is packin'heat; limits of machine learning classifiers based on static analysis features. In *27th Network and Distributed Systems Security (NDSS) Symposium*, 2020.

[2] M. Ahmadi, A. Sami, H. Rahimi, and B. Yadegari. Malware detection by behavioural sequential patterns. *Computer Fraud & Security*, 2013.

[3] B. Alsulami and S. Mancoridis. Behavioral malware classification using convolutional recurrent neural networks. In *13th IEEE International Conference on Malicious and Unwanted Software (MALWARE)*, 2018.

[4] B. Alsulami, A. Srinivasan, H. Dong, and S. Mancoridis. Lightweight behavioral malware detection for windows platforms. In *12th IEEE International Conference on Malicious and Unwanted Software (MALWARE)*, 2017.

[5] F. Breitinger, B. Guttman, M. McCarrin, V. Roussev, and D. White. Approximate matching: definition and terminology. Technical Report NIST SP 800-168, National Institute of Standards and Technology, May 2014.

[6] R. Canzanese, S. Mancoridis, and M. Kam. Run-time classification of malicious processes using system call analysis. In *10th IEEE Conference on Malicious and Unwanted Software (MALWARE)*, 2015.

[7] Q. Chen and R. A. Bridges. Automated behavioral analysis of malware: A case study of wannacry ransomware. In *16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2017.

[8] J. Choi, Y. Han, S.-j. Cho, H. Yoo, J. Woo, M. Park, Y. Song, and L. Chung. A Static Birthmark for MS Windows Applications Using Import Address Table. In *7th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)*, 2013.

[9] A. Dimitriadis, N. Ivezic, B. Kulvatunyou, and I. Mavridis. D4i-digital forensics framework for reviewing and investigating cyber attacks. *Array*, 2020.

[10] A. Duby, T. Taylor, and Y. Zhuang. Malware family classification via residual prefetch artifacts. In *19th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, 2022.

[11] S. Fewer. Reflective dll injection. 2012. *URl: https://github.com/stephenfewer/ReflectiveDLLInjection*.

[12] V. Harichandran, F. Breitinger, and I. Baggili. Bytewise Approximate Matching: The Good, The Bad, and The Unknown. *Journal of Digital Forensics, Security and Law*, 2016.

[13] V. S. Harichandran, D. Walnycky, I. Baggili, and F. Breitinger. Cufa: A more formal definition for digital forensic artifacts. *Digital Investigation*, 2016.

[14] M. Hassen, M. M. Carvalho, and P. K. Chan. Malware classification using static analysis based features. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, 2017.

[15] Í. Íncer Romeo, M. Theodorides, S. Afroz, and D. Wagner. Adversarially robust malware detection using monotonic classification. In *4th ACM International Workshop on Security and Privacy Analytics (IWSPA)*, 2018.

[16] C. Jindal, C. Salls, H. Aghakhani, K. Long, C. Kruegel, and G. Vigna. Neurlux: dynamic malware analysis without feature engineering. In *35th ACM Annual Computer Security Applications Conference (ACSAC)*, 2019.

[17] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro. Transcend: Detecting concept drift in malware classification models. In *26th USENIX Security Symposium*, 2017.

[18] J. Kornblum. Identifying almost identical files using context triggered piecewise hashing. In *6th Annual Digital Forensic Research Workshop (DFRWS)*, 2006.

[19] M. A. Kumara and C. Jaidhar. Leveraging virtual machine introspection with memory forensics to detect and characterize unknown malware using machine learning techniques at hypervisor. *Digital Investigation*, 2017.

[20] A. H. Lashkari, B. Li, T. L. Carrier, and G. Kaur. Volmemlyzer: Volatile memory analyzer for malware classification using feature engineering. In *Reconciling Data Analytics, Automation, Privacy, and Security: A Big Data Challenge (RDAAPS)*. IEEE, 2021.

[21] K. Makhlouf. Finding a needle in a haystack: The traffic analysis version. *Proceedings on Privacy Enhancing Technologies*, 2019.

[22] Mandiant. *Tracking Malware with Import Hashing*, 2014. https://www.fireeye.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html.

[27] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas. Malware classification with recurrent networks. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015.

[23] R. Mosli, R. Li, B. Yuan, and Y. Pan. A behavior-based approach for malware detection. In *IFIP International Conference on Digital Forensics*. Springer, 2017.

[24] M. Narouei, M. Ahmadi, G. Giacinto, H. Takabi, and A. Sami. DLLMiner: Structural mining for malware detection. *Security and Communication Networks*, 2015.

[25] J. Oliver, C. Cheng, and Y. Chen. TLSH–a locality sensitive hash. In *4th IEEE Annual Cybercrime and Trustworthy Computing Workshop (CTC)*, 2013.

[26] F. Pagani, M. Dell'Amico, and D. Balzarotti. Beyond precision and recall: Understanding uses (and misuses) of similarity hashes in binary analysis. In *8th ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2018.

[28] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro. TESSERACT: Eliminating experimental bias in malware classification across space and time. In *28th USENIX Security Symposium*, 2019.

[29] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. In *41st IEEE Symposium on Security and Privacy (S&P)*, 2020.

[30] C. Puodzius, O. Zendra, A. Heuser, and L. Noureddine. Accurate and robust malware analysis through similarity of external calls dependency graphs (ecdg). In *16th International Conference on Availability, Reliability and Security*. IEEE, 2021.

[31] V. Roussev. Data fingerprinting with similarity digests. In *IFIP International Conference on Digital Forensics*. Springer, 2010.

[32] N. Sarantinos, C. Benzaid, O. Arabiat, and A. Al-Nemrat. Forensic Malware Analysis: The Value of Fuzzy Hashing Algorithms in Identifying Similarities. In *2016 IEEE Trustcom/BigDataSE/ISPA*, 2016.

[33] M. Z. Shafiq, S. M. Tabish, F. Mirza, and M. Farooq. PE-Miner: Mining structural information to detect malicious executables in realtime. In *Recent Advances in Intrusion Detection (RAID)*, 2009.

[34] L. F. Sikos. Packet analysis for network forensics: A comprehensive survey. *Forensic Science International: Digital Investigation*, 2020.

[35] J. Stiborek, T. Pevnỳ, and M. Rehák. Multiple instance learning for malware classification. *Expert Systems with Applications*, 2018.

[36] Z. Sun, Z. Rao, J. Chen, R. Xu, D. He, H. Yang, and J. Liu. An opcode sequences analysis method for unknown malware detection. In *2nd International Conference on Geoinformatics and Data Analysis*, 2019.

[37] D. Ucci, L. Aniello, and R. Baldoni. Survey of machine learning techniques for malware analysis. *Computers & Security*, 2019.

[38] J. Upchurch and X. Zhou. Malware provenance: Code reuse detection in malicious software at scale. In *11th IEEE Conference on Malicious and Unwanted Software (MALWARE)*, 2016.

[39] G. D. Webster, B. Kolosnjaji, C. von Pentz, J. Kirsch, Z. D. Hanif, A. Zarras, and C. Eckert. Finding the Needle: A Study of the PE32 Rich Header and Respective Malware Triage. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2017.

[40] G. Wicherski. peHash: A novel approach to fast malware clustering. *2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

[41] S. Yeom, I. Giacomelli, M. Fredrikson, and S. Jha. Privacy risk in machine learning: Analyzing the connection to overfitting. In *31st IEEE Computer Security Foundations Symposium (CSF)*, 2018.