# TagVet: Vetting Malware Tags using Explainable Machine Learning

Lukas Pirch*
Technische Universität
Braunschweig, Germany

Alexander Warnecke*
Technische Universität
Braunschweig, Germany

Christian Wressnegger
Karlsruhe Institute of Technology
Karlsruhe, Germany

Konrad Rieck
Technische Universität
Braunschweig, Germany

## ABSTRACT

When managing large malware collections, it is common practice to use short tags for grouping and organizing samples. For example, collected malware is often tagged according to its origin, family, functionality, or clustering. While these simple tags are essential for keeping abreast of the rapid malware development, they can become disconnected from the actual behavior of the samples and, in the worst case, mislead the analyst. In particular, if tags are automatically assigned, it is often unclear whether they indeed align with the malware functionality. In this paper, we propose a method for vetting tags in malware collections. Our method builds on recent techniques of explainable machine learning, which enable us to automatically link tags to behavioral patterns observed during dynamic analysis. To this end, we train a neural network to classify different tags and trace back its decision to individual system calls and arguments. We empirically evaluate our method on tags for malware functionality, families, and clusterings. Our results demonstrate the utility of this approach and pinpoint interesting relations of malware tags in practice.

## CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**.

## KEYWORDS

Malicious software, dynamic analysis, clustering

**ACM Reference Format:**
Lukas Pirch, Alexander Warnecke, Christian Wressnegger, and Konrad Rieck. 2021. TagVet: Vetting Malware Tags using Explainable Machine Learning. In *14th European Workshop on Systems Security (EuroSec'21), April 26, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3447852.3458719

*Both authors contributed equally to this research.


*EuroSec'21, April 26, 2021, Online, United Kingdom*

ACM ISBN 978-1-4503-8337-0/21/04…$15.00
https://doi.org/10.1145/3447852.3458719

## 1 INTRODUCTION

Despite notable efforts in research, malware still poses a significant threat to computer users. Coping with the sheer amount of new malware variants alone is already a challenging and daunting task. Large security companies, for example, are required to process several hundred thousands of new samples per day [36]. This plethora of malicious code makes it hard to keep abreast of current malware development and update protection mechanisms in time. To alleviate this problem, it is common practice to flag incoming samples with short *tags* that label their origin, format, behavior, or family. These simple tags provide an indispensable tool for maintaining large collections and help focus investigations to particular malware files. As an example, VirusTotal features an extensive search engine for finding such tags in their database of malware samples [37].

Malware tags, however, greatly differ in purpose and quality. While a few tags are manually assigned based on careful reverse engineering, most tags are automatically derived from available information, such as file headers [40, 41], anti-virus labels [31, 32], clusterings [12, 15], and threat intelligence [10, 30]. Although these generated tags provide useful clues for analysis, they may become disconnected from the actual behavior of the malware. For example, tags derived by an imprecise YARA rule [1] might incorrectly flag functionality that is actually not present in the samples. When curating a large collection of malware, it is thus often unclear whether and how automatically generated tags align with the behavior of the samples.

In this paper, we propose TagVet[1], a method for vetting and explaining tags in malware collections. Our method builds on recent techniques of explainable machine learning, which enable us to automatically link tags to behavioral patterns observed during dynamic analysis. To this end, we devise a convolutional neural network for predicting malware tags from monitored system calls and their arguments. By tracing back these predictions through the network, our method determines *why* a tag has been predicted and uncovers its relation to specific system calls. As a result, TagVet links tags to behavioral patterns in retrospective, thereby explaining their semantic relations. Our approach helps to improve the quality of malware collections by exposing errors and inconsistencies in the tagging process even if the process itself is not available.

We empirically evaluate the efficacy of TagVet in a series of experiments with real-world malware and tags for functionality,

[1]Source code will be released at https://github.com/lpirch/tagvet.git

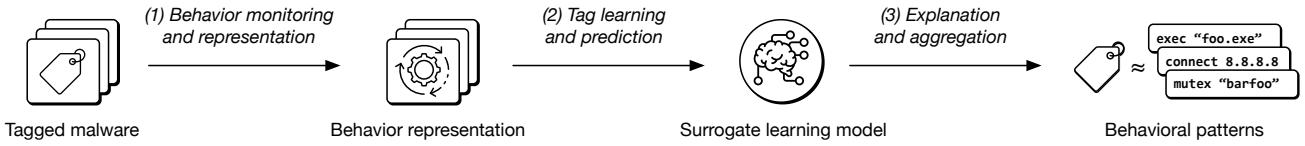


**Figure 1: Schematic depiction of our method TagVet.** (a) Malware is dynamically analyzed, (b) its tags are used to train a surrogate learning model and (c) explanations of the predicted tags are aggregated to behavioral patterns.

families, and clusters. First, we show that our method enables to learn characteristics of tags from malware behavior and predict them with high accuracy. For all considered tag types, TagVet yields an accuracy of 92–97 %. Second, we demonstrate how our method allows for linking tags to system calls and their arguments, thus identifying associated behavioral patterns. For each of the tag types, we quantitatively and qualitatively investigate these behavioral patterns. While for some tags, the patterns align perfectly with our expectations, we also unveil interesting and inconsistent relations of system calls to tags, indicating the need for regularly vetting the tagging process in practice.

## 2 RELATED WORK

Numerous techniques have been proposed for collecting, analyzing, and detecting different forms of malicious code [e.g., 4–6, 17, 18]. Our approach extends this line of research by enabling explanation and vetting of malware tags. Our method TagVet is thus related to work on malware analysis and explainable machine learning, which we briefly discuss in the following.

### 2.1 Malware Analysis and Tags

Techniques for malware analysis can be roughly categorized into *static* and *dynamic* approaches. The former comprise all techniques that inspect malicious code without executing it [e.g., 12, 22, 24, 33]. As static analysis suffers from inherent limitations [see 21], dynamic approaches need to complement them and expose functionality only observable at run-time [e.g., 7, 9, 16, 19, 26]. Several static and dynamic approaches can be used to generate tags for malware automatically. For example, methods for clustering are a common tool for assigning cluster tags to malware files [e.g., 5, 15, 23, 28]. Similarly, information from file headers and metadata provides a valuable source for generating static tags [e.g., 40, 41].

Another branch of malware research has been concerned with deriving family tags from anti-virus labels. Since these labels are notoriously inconsistent among virus scanners, these methods apply different strategies for normalizing and consolidating the labels [e.g., 13, 14, 31, 32]. Recently, this strain of research has been further expanded with methods for deriving tags for general malware categories [32], tagging specific capabilities [25] and predicting functionality using threat intelligence [30].

Despite the breadth of this prior work, however, the retrospective explanation of tags has not been considered in malware research so far. Most tagging methods are black-box systems and opaque to the practitioner. Our approach TagVet addresses this gap and enables to vet tags from static, dynamic, and metadata analysis.

### 2.2 Explainable Machine Learning

TagVet rests on recent techniques of explainable machine learning. These techniques aim at explaining how a learning model arrives at its decisions. To achieve this, a *relevance map* is derived that indicates how each input feature contributes to a decision. These maps are typically determined using gradients [e.g., 3, 34, 35] or approximations of the learning model [e.g., 11, 20, 27]. For a general discussion of explanation methods in computer security, we refer the reader to the overview article by Warnecke et al. [39].

For our method, we employ *layer-wise relevance propagation (LRP)* [3], an effective and general-purpose approach for explaining neural networks and other learning models. Note that LRP as well as all other methods for explainable learning inherit the weaknesses of the underlying learning models and thus are vulnerable to adversarial examples [8, 42]. In the context of our approach, however, such attacks play a minor role, as there exist more potent means for evading malware analysis in general.

## 3 APPROACH

Our method builds on the three analysis phases shown in Figure 1. First, it executes tagged malware samples in a sandboxed environment and encodes the monitored behavior in a way suitable for analysis (Section 3.1). Second, it trains a convolutional neural network as a surrogate model for the tagging process, such that tags can be predicted from monitored behavior (Section 3.2). Finally, our method uses layer-wise relevance propagation to explain the network's predictions and link the tags back to patterns in the monitored behavior (Section 3.3).

### 3.1 Behavior Monitoring and Representation

In the first stage, malware samples are executed in a sandboxed environment to monitor their behavior. We employ *VMRay Analyzer*, a hypervisor-based sandbox for the Windows platform [38] that records all system calls to operating system libraries and the kernel. This dynamic analysis yields a behavior report for each executed sample that comprises lists (threads) of system calls with respective arguments. Our analysis hence operates on the boundary of the operating system and characterizes malware through its interaction with the host API.

*Consolidation.* As the sandbox reports contain very fine-granular data from the operating system state, we apply different consolidation steps before analyzing the behavior with a neural network. First, volatile call arguments, such as memory addresses and process identifiers, depend only on the system state and can safely be ignored. Second, we observe several function call arguments

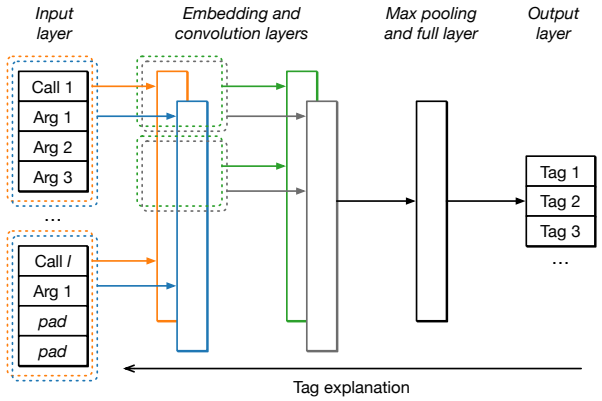


**Figure 2: CNN for tag prediction:** The first convolution captures system calls with their arguments, the second convolution summarizes multiple calls and the last layer returns tag probabilities.

consisting of rare substrings, such as temporary file names. To consolidate these, we split each argument using appropriate delimiters ("\" for file and registry paths and "." for network addresses) and analyze the frequency of the resulting substrings. Substrings appearing in less than 10 % of the reports are then replaced with a wildcard symbol "⋆". In our evaluation, this replacement reduces the number of distinct substrings by 99.7 %, while still preserving most information relevant to the analyst.

*Representation of behavior.* To achieve a unified representation of the behavior described in the sandbox reports, we consider each report as a sequence of system calls and their arguments. For simplicity, we ignore the relation of threads in this representation, as the convolutional neural network used in TagVet can focus on local patterns in the data. Formally, we map a report to a sequence $x = (x_1, \dots x_l)$ of $l$ tokens, where each token either corresponds to a system call or an argument. As the number of arguments varies between calls, we pad all arguments to a fixed number $n$ using a special *pad* token. Consequently, if the report contains $s$ system calls, the final sequence $x$ has a length of $l = s \cdot (1 + n)$.

## 3.2 Tag Learning and Prediction

For the next stage of TagVet, we require a machine-learning model that predicts tags for a given behavior. We refer to this model as a *surrogate model* as it mimics the original tagging process. While several approaches might be applicable for learning to predict tags, we employ *convolutional neural networks (CNNs)* that can be precisely tailored to the problem at hand.

*Convolutional neural networks.* Generally, a CNN maps a sequence $x$ to a vector $z$ where each entry $z_i$ corresponds to the probability that $x$ belongs to class $c_i$. To this end, a CNN uses multiple convolutional kernels $K_1, \dots, K_p$ where each kernel consists of a weight vector, which are optimized during training as the parameters of the network. By design, CNNs extract local patterns from data, making them a suitable choice for classification tasks over sequences.

*Convolution of system calls.* We design the CNN for our approach *explicitly* to process the encoded malware behavior. The input to the network consists of a padded sequence of tokens reflecting system calls and arguments. Hence, we compose the first convolutional layer of $m_1$ different filters of size $n + 1$ which are slid over the input, such that they process a complete system call with its argument at a time (Figure 2, left). To provide vector inputs for this convolution, we pass the tokens through an embedding layer which is also learned during training. The second convolutional layer has $m_2$ filters and performs convolutions on the output of the first layer (Figure 2, middle). Hence, this layer infers dependencies between different system calls and captures broader patterns in malware behavior. Since the size of this convolution depends on the input length we employ a max-pooling layer that maps the output of the second convolutional layer to a vector of size $m_2$. This vector is finally fed to a fully-connected layer that returns probabilities for the tags of the input sample (Figure 2, right).

## 3.3 Generating Explanations

Once a surrogate model has been trained, we are able to apply techniques of explainable machine learning to interpret its prediction process and unveil behavior associated with the tags. In particular, our method TagVet assigns each token $x_i$ of the sequence $x$ a relevance score $R_i$, indicating its importance for predicting a tag. As the entries correspond to system calls and arguments, this process enables us to pinpoint behavior relevant for specific tags.

*Layerwise relevance propagation.* Following the recommendations of Warnecke et al. [39], we use LRP [3] to compute the relevance scores in our approach. LRP performs a backwards-pass through a neural network from the output to the input, such that the following conservation property holds,

$$\sum_i R_i^1 = \sum_i R_i^2 = \cdots = \sum_i R_i^L \tag{1}$$

where $R_i^j$ denotes the relevance score assigned to the $i$-th unit in the $j$-th layer. This property ensures that the output score of the network is completely transferred to the input and results in concise explanations [2]. The sign of $R_i^j$ can be either positive (speaking *for* the classification) or negative (speaking *against* it). In the remainder of this paper, we thus normalize the relevance scores to lie in the interval $[-1, 1]$.

As an example, Table 1 shows a simplified snippet of an explanation generated for the behavior tag *"creates process with hidden window"* assigned by the VMRay Analyzer (see Section 4). The relevance scores are indicated by blue shading. The system call argument `create_suspended` obtains the highest relevance, as it is typically used to create a process for a background window. Other highlighted tokens, e.g. `sw_hide` or `show_window`, also fit perfectly to the tag, indicating that it matches the defined behavior well.

*Aggregating behavioral patterns.* The tokens with the strongest influence alone can be meaningless without the surrounding context. Imagine, for example, that the most relevant token for a tag corresponds to the argument "`true`". As a remedy, we propose an aggregation scheme for the explanations of TagVet, tailored to the context of program behavior. To represent the context of a token,

**Table 1: Explanation snippet for behavior tag "Creates process with hidden window".** Relevance is shown by blue shading.

| Id | Token Name | Id | Token Name |
|---|---|---|---|
| 0 | proc_create | | |
| 1 | symbol_name | 2 | createprocess |
| 3 | creation_flags | 4 | create_suspended |
| 5 | show_window | 6 | sw_hide |
| 7 | success | 8 | true |

we use a notation inspired by the Python programming language that builds on named arguments.

For each relevant token, we identify the related system call and then compose an explanation describing this call with a named argument. As an example, for the snippet in Table 1, we write

`proc_create(in:creation_flags="create_suspended")`,

indicating that the relevant token `create_suspended` belongs to the system call `proc_create`. The prefix `in:` denotes an input argument whereas `out:` signifies a return value. To generate a behavioral pattern from such calls, we compute the surroundings of the 10 most relevant tokens in every sample and count their occurrences in the entire dataset. Then, we average the relevance values of the single aggregations and sort them by their occurrence in descending order to obtain behavioral patterns.

## 4 EVALUATION

We proceed to evaluate the effectivity of TagVet in a series of experiments with real-world malware. In particular, we explore how our method learns to predict tags and whether its explanations help to understand the underlying malware behavior. To this end, we first present a quantitative evaluation of our approach (Section 4.2) and then qualitatively discuss four case studies (Section 4.3).

### 4.1 Experimental Setup

We collect malware from the VirusShare repository [29] and focus on samples that target the Windows platform. In particular, we retrieve a recent subset of 65,536 samples and extract all valid PE files, resulting in 6,598 malware samples. Each of these samples is then labeled by multiple virus scanners and we use AVClass [31] to determine their family labels. As we intend to simulate the vetting process in practice, we filter out very small families with less than 10 samples, resulting in 5,217 malware samples from 71 families. Finally, we execute each sample in the VMRay Analyzer sandbox [38] with simulated user traces and internet connectivity to monitor as much malicious behavior as possible.

*Malware tags.* We consider three types of tags for our experiments: First, we use the family labels assigned by AVClass as *family tags*. Second, the VMRay Analyzer comes with 60 predefined threat indicators that are matched during monitoring and result in *sandbox tags*, such as *"creates process with hidden window"*. Third, we conduct a behavior-based clustering similar to Rabadi and Teo [26]. In particular, we automatically group the samples using complete-linkage clustering based on tuples of system calls and arguments.

To find a high-quality clustering, we evaluate the parameter space and use the elbow method to determine peaks in the silhouette score for determining the optimal number of clusters. Figure 3 shows the mean Silhouette Coefficient and the Adjusted Rand Index for different number of clusters, and the chosen configuration as dashed line at 30 clusters.

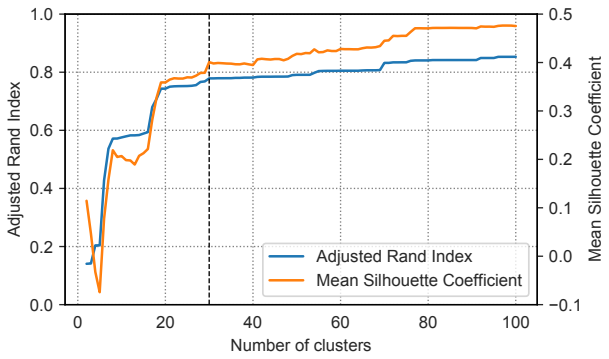


**Figure 3: Mean Silhouette Coefficient and Adjusted Rand Index for different clusterings.**

*Setup of TagVet.* After consolidation, our dataset comprises 5,217 sequences with 4,241 unique tokens. We train the CNN model on this data using an embedding dimension of 128 and a fixed number of 21 system call arguments. The filter sizes of the two convolutional layers are 21 and 5, respectively, and we use 64 filters for each layer. The model is trained with the TensorFlow library using the Adam optimizer and the categorical cross-entropy loss function.

### 4.2 Quantitative Evaluation

We start our evaluation by first investigating the prediction performance of TagVet and the quality of the generated explanations. For these experiments, we split our dataset into a training, validation, and test partition using 80 %, 10 %, and 10 % of the data, respectively. We train the CNN on the training data and use the validation partition to calibrate all model parameters. The final model is then applied to the (unseen) test data. This procedure is repeated five times, and the performance is averaged.

*Prediction performance.* We measure the prediction performance of TagVet using three standard metrics: First, we use the *accuracy* to determine how many samples are correctly classified. Next, we measure the *area under the ROC curve*, which describes the relation between true-positive and false-positive predictions. Finally, we report the *area under precision-recall curve* that provides a view on the precision of our approach. Since these metrics are designed for binary classification, we use them on every tag class separately and compute a weighted average. That is, tag classes that occur often have a proportionally higher weight.

Table 2 shows the prediction performance of TagVet. We observe that the CNN can successfully predict the different tags for all types with high accuracy. The best results are achieved for the sandbox tags, likely because these are directly derived from the behavior monitored in the sandbox. The family and clustering tags,

**Table 2: Prediction performance of TagVet.** All performance metrics are averaged over the tag classes.

| Tag Class | # Tags | Accuracy | AUC (ROC) | AUC (PR) |
|---|---|---|---|---|
| Sandbox | 60 | 0.97 ± 0.002 | 0.99 ± 0.001 | 0.98 ± 0.002 |
| Family | 71 | 0.94 ± 0.007 | 0.96 ± 0.007 | 0.85 ± 0.015 |
| Clustering | 30 | 0.92 ± 0.019 | 0.99 ± 0.004 | 0.95 ± 0.013 |

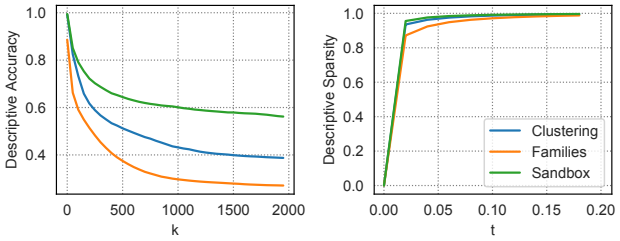


**Figure 4: Descriptive accuracy and sparsity of TagVet.**

on the other hand, also yield a strong accuracy. The overall performance of the CNN is excellent and justifies the usage of explainable learning for vetting tags in our approach.

*Explanation quality.* To evaluate the explanations generated by TagVet, we use the *descriptive accuracy (DA)* and the *descriptive sparsity (DS)* [39]. The DA measures how well an explanation reflects features contributing to a prediction. Given the predicted class $c_i$, the $k$ most relevant features $x_1, \ldots, x_k$, and a constant replacement value $\epsilon$, this measure is defined as

$$\mathrm{DA}_k(x, f) = f(x_1 = \epsilon, \ldots, x_k = \epsilon)_{c_i}, \tag{2}$$

that is, it returns the classification score without the $k$ most relevant features. With increasing $k$, a good explanation will quickly drop in descriptive accuracy, as the relevant features are also highly important for obtaining a correct prediction.

The descriptive sparsity complements the descriptive accuracy by measuring how many values of the explanation are close to zero and thus can be ignored. Concretely, the sparsity computes the *mass around zero (MAZ)*, given by

$$\mathrm{MAZ}(t) = \int_{-t}^{t} h(x)\mathrm{d}x, \qquad t \in [0, 1] \tag{3}$$

where $h$ is the normalized histogram of the relevance map. A good explanation reaches a high sparsity at a small value of $t$, that is, only very few features receive a high relevance and thus the explanation can be better interpreted by a human analyst.

Figure 4 shows the values of DA and DS for all tag classes, where we choose the *pad* token for $\epsilon$ when computing DA. The DA score drops quickly for all tag classes when the most important features are removed. For example, removing the top 50 features reduces the accuracy by 16.7 %, 22.4 % and 14.4 % for the clustering, family and sandbox tags. Considering the large number of different tags and that features may only be important for a fraction of the classes, this result indicates a high quality of the explanations. Moreover, we find that the descent in the curve for the sandbox tags is not as steep as for the other ones. We conjecture that this effect stems from the more complex tagging process, as one sample can be assigned multiple sandbox tags. For the DS score, we observe that all curves reach high values at $t = 0.02$. Hence, more than 90 % of the features are irrelevant and only a few important ones form the explanations for the different tags.

## 4.3 Qualitative Evaluation

The previous experiments demonstrate the quality of the learned model, yet the generated explanations need to also be sensible from an analyst's perspective. We therefore perform four qualitative case studies to verify the semantic coherence of the extracted patterns with human expectations.

*Sandbox tags.* As the first case study, we inspect a random sample of behavioral patterns for the 60 sandbox tags. We observe that all patterns align well with the names of the underlying threat indicators and there is a clear relation between behavior and the tag semantics. Table 3 shows an example for the tag *"attempts to connect to unavailable TCP servers"*. As expected, the tag is supported by the socket connection function (line 1) in combination with the unsuccessful invocation attribute (line 3), resulting in a concise summary of the threat indicator matched by the sandbox.

**Table 3: Behavioral pattern for the sandbox tag "Attempts to connect to unavailable TCP servers".**

| Id | Tokens in context |
|---|---|
| 1 | `sck_connect(*)` |
| 2 | `sck_connect(in:post_symbol_name="connect")` |
| 3 | `sck_connect(out:success="false")` |
| 4 | `sck_connect(in:remote_port="*")` |

*Family tags.* Regarding the malware families, we observe much more specificity in the explanations. This makes sense because generic behavioral patterns such as accessing external IP addresses are occurring across different families and, therefore, are not useful for their explanation. In this case study, we examine the *Fareit* malware family, for which Table 4 shows the behavior explanation. We find that creating a window with `windprocparameter=0` as argument appears among the top ten most relevant features in 79 % of the explanations of the family's samples. Also, 78 % of the *Fareit* samples sleep for exactly 25 s, whereas instances from the *Autoit* malware family, for example, typically sleep only for 750 ms. Judging by the steep drop in accuracy when removing these features (see Figure 4), we conclude that the rather detailed behavioral patterns are characteristic for specific malware families and are indeed suitable candidates for behavioral detection rules.

*Cluster tags.* Third, we examine the explanations for cluster tags. When analyzing a random sample of the 71 cluster tags, we observe that several of them strongly overlap with family tags, which indicates a successful clustering procedure. However, a few behavioral patterns also deviate from the respective families. As an example, Table 5 shows the output for cluster #15, which contains system calls for reading out environment variables and sending an HTTP request to "`ipv4bot.whatismyipaddress.com`". Interestingly, the presented tokens are not identical to the ones from the respective

**Table 4: Behavioral pattern for the malware family Fareit.**

| Id | Tokens in context |
|---|---|
| 1 | wnd_create(in:wndproc_parameter="0") |
| 2 | wnd_create(in:width="320") |
| 3 | sleep(in:milliseconds="25000") |
| 4 | sleep(out:milliseconds_text= "25000 milliseconds (25.000 seconds)") |
| 5 | wnd_create(in:class_name="t__304124810") |

malware family but correspond to it semantically: The *Gandcrab* malware scans the user environment and determines its IP address when executed. Furthermore, we randomly select 10 reports from that cluster to see whether the wildcard in the file system path conceals relevant information. This is not the case as the respective function call occurs but the wildcard only replaces an MD5 hash value in all of the inspected files.

**Table 5: Behavioral pattern for the cluster #15.**

| Id | Tokens in context |
|---|---|
| 1 | env_get(in:symbol_name="getenvironmentvariable") |
| 2 | file_create(in:file_name_orig="c:\users \%USERNAME%\desktop*.exe") |
| 3 | str_len(in:string="pridur") |
| 4 | file_create(in:file_name="c:\users \%USERNAME%\desktop*.exe") |
| 5 | open_http_request(in:url= "ipv4bot.whatismyipaddress.com/*") |
| 6 | open_connection(in:server= "ipv4bot.whatismyipaddress.com") |

*Tagging inconsistencies.* In addition, we investigate how tag explanations enable human analysts to reason about the quality of tags. In this final case study, we examine tagging inconsistencies between the family and behavior tags. That is, finding cases in which for example one family tag is split across multiple behavioral clusters or vice-versa. Analyzing the explanations from the CNN operating on behavior reports can then give evidence about the discrepancies between behavioral tags and the ones generated using different analyses. For instance, anti-virus products create family tags based on various sources of information, including file metadata and static analysis. Hence, we expect some family explanations to contain static artifacts when malware cannot be discriminated from a behavioral perspective alone.

We find this to be the case for the *AutoIt* family. The name refers to a Windows scipting language which suggests that the family tagging process is likely based on malware being written in that language. However, this would not be directly observable in behavioral reports which is confirmed by the following two findings. First, the *Autoit* family explanation in Table 6 shows that the CNN relies on an artifact for correct classification: The window creation function in line 5 contains the family name in one of its arguments. Most interestingly, this artifact is of high relevance when training the CNN to predict family tags but never occurs in explanations for behavior tags. A second finding is that the 136 *AutoIt* samples are scattered across 15 behavioral clusters. The explanation for the cluster with the largest number of *AutoIt* samples (44/136) indeed contains some related behavior. According to the cluster explanation, loading the functions "VarSub", "VarMod" and "VarDiv" from a dynamically linked library accounts for three of the top five most relevant features and is present in more than 99 % of the explanations. These functions are used for basic arithmetic operations, are part of the Windows API (oleauto.h) and are internally used by the *AutoIt* language in version 3 which coincides with the artifact observed in Table 6. Yet, the functions are not exclusively available to this scripting engine and hence might be used by other malware families as well. Furthermore, the *AutoIt* samples comprise only 14.8 % of all samples in the analyzed cluster and explanations for other clusters with *AutoIt* samples contain no relatable information at all. We conclude from these weak behavioral indicators and the lack of explanation coherence between this family and the related clusters that TagVet is able to give evidence for reasoning about whether certain family tags are sensible from a behavioral perspective.

**Table 6: Behavioral pattern for the malware family AutoIt.**

| Id | Tokens in context |
|---|---|
| 1 | sys_sleep(in:milliseconds=750) |
| 2 | sys_sleep(out:milliseconds_text= "750 milliseconds (0.750 seconds)") |
| 3 | wnd_create(in:wnd_proc_parameter=0) |
| 4 | sys_sleep(in:post_symbol_name="settimer") |
| 5 | wnd_create(in:window_name="autoit v3") |

**Table 7: Behavioral pattern for the cluster #28.**

| Id | Tokens in context |
|---|---|
| 1 | wnd_create(in:wnd_proc_parameter=0) |
| 2 | mod_get_proc_address(in:module_name= "c:\windows\syswow64\user32.dll") |
| 3 | mod_get_proc_address(in:function="VarSub") |
| 4 | mod_get_proc_address(in:function="VarMod") |
| 5 | mod_get_proc_address(in:function="VarDiv") |

Our findings demonstrate the utility of our approach: Depending on the particular tags (sandbox, family, or clustering), only those features are identified that are relevant in the particular context. As a result, a cluster only partially overlapping with a malware family yields a different behavioral pattern and thus helps to understand what characteristics the tags reflect.

## 5 CONCLUSION

The ever-increasing amount of malware instances and novel strains of malicious code calls for effective strategies for organizing newly discovered samples. Simple tags can be a powerful tool to quickly categorize and sort new variants. The efficacy of this strategy, however, hinges on the quality of the assigned tags. With TagVet, we provide a method for vetting tags in malware collections. Our

method enables to unveil the malware behavior associated with a tag and allows an analyst to recognize inconsistencies in the tagging process. We demonstrate the utility of this approach for different types of tags used in day-to-day malware analysis. Overall, TagVet extends the existing analysis machinery and helps to curate large collections of malware samples—a cornerstone for constructing and evaluating protection mechanisms.

## ACKNOWLEDGMENTS

The authors gratefully acknowledge funding from the German Federal Ministry of Education and Research (BMBF) under the projects VAMOS (FKZ 16KIS0534) and IVAN (FKZ 16KIS1167). Furthermore, we acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy EXC 2092 CASA-390781972 and by the Ministerium für Wirtschaft, Arbeit und Wohnungsbau Baden-Wuerttemberg under the project Poison-Ivy.

## REFERENCES

[1] V. M. Alvarez. Yara – the pattern matching swiss knife for malware researchers. https://virustotal.github.io/yara/. visited March 2021.
[2] M. Ancona, E. Ceolini, C. Öztireli, and M. Gross. Towards better understanding of gradient-based attribution methods for deep neural networks. In *Proc. of International Conference on Learning Representations (ICLR)*, 2018.
[3] S. Bach, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, and W. Samek. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PLoS ONE*, 10(7), July 2015.
[4] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of internet malware. In *Proc. of International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 178–197, 2007.
[5] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2009.
[6] J. Caballero, C. Grier, C. Kreibich, and V. Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *Proc. of USENIX Security Symposium*, 2011.
[7] A. Dinaburg, P. Royal, M. I. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, pages 51–62, 2008.
[8] A.-K. Dombrowski, M. Alber, C. J. Anders, M. Ackermann, K.-R. Müller, and P. Kessel. Explanations can be manipulated and geometry is to blame. In *Advances in Neural Information Proccessing Systems (NIPS)*, 2019.
[9] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2): 1–42, 2012.
[10] H. Gascon, B. Grobauer, T. Schreck, L. Rist, D. Arp, and K. Rieck. Mining attributed graphs for threat intelligence. In *Proc. of the ACM Conference on Data and Applications Security and Privacy (CODASPY)*, pages 15–22, Mar. 2017.
[11] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing. Lemna: Explaining deep learning based security applications. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, pages 364–379, 2018.
[12] X. Hu, S. Bhatkar, K. Griffin, and K. G. Shin. Mutantx-s: Scalable malware clustering based on static feature. In *Proc. of USENIX Annual Technical Conference*, pages 187–198, 2013.
[13] M. Hurier, K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon. On the lack of consensus in anti-virus decisions: Metrics and insights on building ground truths of android malware. In *Proc. of International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 142–162, 2016.
[14] M. Hurier, G. Suarez-Tangil, S. K. Dash, T. F. Bissyandé, Y. Le Traon, J. Klein, and L. Cavallaro. Euphony: Harmonious unification of cacophonous anti-virus vendor labels for Android malware. In *Proc. of International Conference on Mining Software Repositories (MSR)*, pages 425–435, 2017.
[15] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, pages 309–320, 2011.
[16] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. yong Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proc. of USENIX Security Symposium*, pages 351–366, 2009.
[17] C. Kolbitsch, B. Livshits, B. G. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *Proc. of IEEE Symposium on Security and Privacy*, pages 443–457, 2012.
[18] P. Kotzias, L. Bilge, and J. Caballero. Measuring pup prevalence and pup distribution through pay-per-install services. In *Proc. of USENIX Security Symposium*, pages 739–756, 2016.
[19] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti. Detecting environment-sensitive malware. In *Proc. of International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 338–357, 2011.
[20] S. M. Lundberg and S.-I. Lee. A unified approach to interpreting model predictions. In *Advances in Neu- ral Information Proccessing Systems (NeurIPS)*, pages 4765–4774, 2017.
[21] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, pages 421–430, 2007.
[22] M. Neugschwandtner, P. M. Comparetti, G. Jacob, and C. Kruegel. Forecast: skimming off the malware cream. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, pages 11–20, 2011.
[23] R. Perdisci and M. U. Vamo: towards a fully automated malware clustering validity analysis. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, pages 329–338, 2012.
[24] R. Perdisci, A. Lanzi, and W. Lee. Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, pages 301–310, 2008.
[25] J. Qiu, J. Zhang, W. Luo, L. Pan, S. Nepal, Y. Wang, and Y. Xiang. A3cm: Automatic capability annotation for Android malware. *IEEE Access*, 7:147156–147168, 2019.
[26] D. Rabadi and S. G. Teo. Advanced windows methods on malware detection and classification. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, pages 54–68, 2020.
[27] M. T. Ribeiro, S. Singh, and C. Guestrin. "why should i trust you?": Explaining the predictions of any classifier. In *Proc. of ACM SIGKDD International Conference On Knowledge Discovery and Data Mining (KDD)*, 2016.
[28] K. Rieck, P. Trinius, C. Willems, and T. Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security (JCS)*, 19(4): 639–668, June 2011.
[29] J.-M. Roberts. Virusshare.com. https://www.virusshare.com. visited March 2021.
[30] E. M. Rudd, F. N. Ducau, C. Wild, K. Berlin, and R. E. Harang. Aloha: Auxiliary loss optimization for hypothesis augmentation. In *Proc. of USENIX Security Symposium*, pages 303–320, 2019.
[31] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero. Avclass: A tool for massive malware labeling. In *Proc. of International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 230–253, 2016.
[32] S. Sebastián and J. Caballero. Avclass2: Massive malware tag extraction from av labels. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, pages 42–53, 2020.
[33] M. I. Sharif, V. Yegneswaran, H. Saïdi, P. A. Porras, and W. Lee. Eureka: A framework for enabling static malware analysis. In *Proc. of European Symposium on Research in Computer Security (ESORICS)*, pages 481–500, 2008.
[34] K. Simonyan, A. Vedaldi, and A. Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. In *Proc. of International Conference on Learning Representations (ICLR)*, 2014.
[35] M. Sundararajan, A. Taly, and Q. Yan. Axiomatic attribution for deep networks. In *Proc. of International Conference on Machine Learning (ICML)*, pages 3319–3328, 2017.
[36] X. Ugarte-Pedrero, M. Graziano, and D. Balzarotti. A close look at a daily dataset of malware samples. *ACM Transactions on Privacy and Security*, 22(1), 2019.
[37] VirusTotal. Vt intelligence: Combine Google and Facebook and apply it to the field of malware. https://www.virustotal.com/gui/intelligence-overview, visited March 2021.
[38] VMRay GmbH. Malware analysis sandbox & malware detection software. https://www.vmray.com/products/analyzer-malware-sandbox/. visited March 2021.
[39] A. Warnecke, D. Arp, C. Wressnegger, and K. Rieck. Evaluating explanation methods for deep learning in computer security. In *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*, Sept. 2020.
[40] 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 *Proc. of International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 119–138, 2017.
[41] G. Wicherski. peHash: A novel approach to fast malware clustering. In *Proc. of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.
[42] X. Zhang, N. Wang, H. Shen, S. Ji, X. Luo, and T. Wang. Interpretable deep learning under fire. In *Proc. of USENIX Security Symposium*, pages 1659–1676, 2020.