AVClass2: Massive Malware Tag Extraction from AV Labels

Silvia Sebastián
IMDEA Software Institute
Universidad Politécnica de Madrid
silvia.sebastian@imdea.org

Juan Caballero
IMDEA Software Institute
Madrid, Spain
juan.caballero@imdea.org

ABSTRACT

Tags can be used by malware repositories and analysis services to enable searches for samples of interest across different dimensions. Automatically extracting tags from AV labels is an efficient approach to categorize and index massive amounts of samples. Recent tools like AVClass and Euphony have demonstrated that, despite their noisy nature, it is possible to extract family names from AV labels. However, beyond the family name, AV labels contain much valuable information such as malware classes, file properties, and behaviors.

This work presents AVClass2, an automatic malware tagging tool that given the AV labels for a potentially massive number of samples, extracts clean tags that categorize the samples. AVClass2 uses, and helps building, an *open taxonomy* that organizes concepts in AV labels, but is not constrained to a predefined set of tags. To keep itself updated as AV vendors introduce new tags, it provides an update module that automatically identifies new taxonomy entries, as well as tagging and expansion rules that capture relations between tags. We have evaluated AVClass2 on 42M samples and showed how it enables advanced malware searches and to maintain an updated knowledge base of malware concepts in AV labels.

CCS CONCEPTS

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

KEYWORDS

AV Labels; Tag; Malware; Taxonomy

ACM Reference Format:
Silvia Sebastián and Juan Caballero. 2020. AVClass2: Massive Malware Tag Extraction from AV Labels. In *Annual Computer Security Applications Conference (ACSAC 2020), December 7–11, 2020, Austin, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3427228.3427261

1 INTRODUCTION

Tags are keywords assigned to data objects (e.g., documents, videos, images) to categorize them and to enable efficient searches along different dimensions such as properties, ownership, and origin. Tags can be used by malware repositories and malware analysis services (e.g., [16, 36]) for enabling searches for samples of interest.

Malware tags can be manually produced by analysts, e.g., during reverse engineering of a malware sample, or output by analysis tools such as sandboxes and signature-matching engines. In both cases, each analyst and tool developer may use its own *vocabulary*, i.e., their own custom set of tags. This is similar to user tagging, or *folksonomies*, in Web services [11, 18], which are known to lead to issues such as tags produced by different entities being *aliases* (or synonyms) for the same concept, some tags being highly specific to the entity producing them, and a tag from an entity corresponding to multiple tags from another entity. To address these issues, standards such as Malware Attribute Enumeration and Characterization (MAEC) [24] define a language for sharing malware analysis results. However, they have low adoption due to their use of rigid *controlled vocabularies* (i.e., predefined tags) that may not always match analyst needs, require frequent updates, and are necessarily incomplete, e.g., MAEC does not include malware families.

Detection labels by anti-virus engines (i.e., *AV labels*) are an instance of the above problem. An AV label can be seen as a serialization of the tags an AV engine assigns to the sample. Since tags are selected by each AV vendor rather independently, inconsistencies among labels from different vendors are widespread, as frequently observed in malware family names [2, 5, 9, 25, 28, 35]. Recent tools like AVClass [35] and Euphony [9] demonstrate that, despite their noisy nature, it is possible to extract accurate family tags from AV labels. However, beyond the family name, AV labels may also contain much valuable information such as the class of malware (e.g., *ransomware*, *downloader*, *adware*), file properties (e.g., *packed*, *themida*, *bundle*, *nsis*) and behaviors (e.g., *spam*, *ddos*, *infosteal*).

Automatically extracting malware tags from AV labels enables to cheaply categorize and index massive amounts of samples without waiting for those samples to be statically analyzed or executed in a sandbox. And, since different AV vendors may execute a sample, the extracted tags may accumulate behaviors observed under different conditions. Furthermore, AV labels may encode domain knowledge from human analysts that is not produced by automated tools. Once obtained, the tags can be used to enable efficient search of samples of a specific class, type, family, or with a specific behavior. And, the identified samples can then be used as ground truth for machine learning approaches [1, 3, 31, 34].

Unfortunately, extracting useful tags from AV labels is challenging due to different vocabularies used by AV engines. For example, Engine1 may include *multiplug* in its label for Sample1, while Engine2 includes *brappware* in its label for Sample2, which is not detected by Engine1. Both tags refer to samples that modify the browser by installing plugins. Thus, when an analyst searches for behavior *browsermodify*, both samples should appear as results.

ACSAC 2020, December 7–11, 2020, Austin, USA

ACM ISBN 978-1-4503-8858-0/20/12…$15.00
https://doi.org/10.1145/3427228.3427261

This work presents AVclass2, an automatic malware tagging tool that given the AV labels for a potentially massive number of samples, extracts for each input sample a clean set of tags that capture properties such as the malware class, family, file properties, and behaviors. AVclass2 ships with a default *open taxonomy* that classifies concepts in AV labels into *categories*, as well as default *tagging rules* and *expansion rules* that capture relations between tags. In contrast to closed taxonomies, AVclass2 does not mandate a predefined set of tags. Instead, unknown tags in AV labels, e.g., a new behavior or family name, are also considered. AVclass2 has an *update module* that uses tag co-occurrence to identify relations between tags. Those relations are a form of generalized knowledge that the update module uses to automatically generate taxonomy updates, tagging, and expansion rules to keep the tool updated as AV vendors introduce new tags. Thus, AVclass2 can maintain an updated knowledge base of malware concepts in AV labels.

AVclass2 builds on AVclass [35]. The goal is to perform the minimum needed changes to AVclass that allow it to evolve from a malware labeling tool, focused exclusively on malware families, to a malware tag extraction tool that provides rich threat intelligence by extracting and structuring all useful information in AV labels. Thus, AVClass2 inherits AVClass major properties: scalability, AV engine independence, platform-agnostic, no access to samples required (only to their labels), and open source.

We have evaluated AVclass2 on 42M samples and compared it with AVclass and Euphony, the two state-of-the-art malware family labeling tools. We show how the tags AVclass2 extract enable rich searches on malware samples, not possibly with existing tools, how the extracted tags are complementary to those already in use by popular malware repositories such as VirusTotal [36], and how the update module can be used to maintain an updated knowledge base of malware concepts in AV labels.

The main properties of AVclass2 are:

- Automatically extracts tags from AV labels that categorize malware samples according to their malware class, family, behaviors, and file properties.
- Uses and builds an open taxonomy that does not use a closed set of tags, and thus can handle new tags introduced over time by AV vendors.
- Can expand the input taxonomy, tagging rules, and expansion rules, by generalizing relations found in AV labels, allowing to maintain over time an up-to-date knowledge base of malware concepts in AV labels.
- Evaluated on 42M samples and compared with the two state-of-the-art malware family labeling tools [9, 35].
- Open source[1].

2 OVERVIEW

When an AV engine flags a sample as malicious it returns an *AV label* (or simply a *label* for short), i.e., a string with information about the malicious sample. An AV label can be seen as a sequence of tags separated by delimiters. Those tags are engine-specific as they are selected rather independently by the AV vendor. For clarity, we will use *tokens* to refer to engine-specific tags in AV labels and *tags* to refer to entries in the taxonomy. Some tokens in AV labels contain useful information such as the malware class (e.g., *virus*, *worm*), malware family (e.g., *zbot*, *virut*), file properties (e.g., *pdf*, *toolbar*), and behaviors usually retrieved dynamically (e.g., *ddos*, *proxy*). Other tokens do not provide useful information because they are generic (e.g., *malicious*, *application*) or very specific to the AV vendor, and thus of little use for external analysts, such as the detection technology (e.g., *deepscan*, *cloud*) or the specific variant of a malware family (e.g., *aghr*, *bcx*). At a high level, the goal of AVclass2 is to distinguish the tokens that provide useful information, identify relations between tokens from different AV engines, and convert them into tags in the taxonomy.

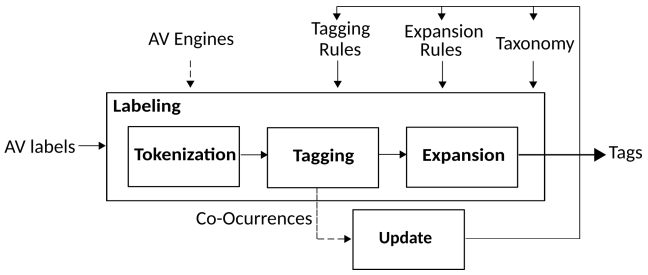

Figure 1: AVclass2 **architecture.**

The architecture of AVclass2 is shown in Figure 1. It comprises of two modules: *labeling* and *update*. The labeling module takes as input the AV labels assigned by multiple AV engines to the same samples, an optional list of AV engines whose labels to use (if not provided, all AV labels are used), a set of tagging rules, an optional set of expansion rules, and a taxonomy that classifies tags into categories and captures parent-child relationships between tags in the same category. For each input sample, it outputs a set of tags ranked by the number of AV engines including the tag's concept in their label. AVclass2 ships with default tagging rules, default expansion rules, and a default taxonomy. Thus, AVclass2 can be used out-of-the-box without the need for any configuration. However, AVclass2 is fully configurable, so the analyst can easily plug-in its own tagging rules, expansion rules, and taxonomy.

The update module takes as input the co-occurrence statistics, tagging rules, expansion rules, and taxonomy. It first identifies strong relations between tags, which generalize knowledge beyond individual samples, e.g., that a family is ransomware or sends SMS. Then, it uses inference rules on the relations to automatically propose new tagging rules, new expansion rules, and taxonomy updates, which are then fed back to the labeling.

The remainder of this section introduces the labeling module in Section 2.1, describes the input taxonomy in Section 2.2, and introduces the update module in Section 2.3.

2.1 Labeling Module Overview

The labeling module comprises of three steps: tokenization, tagging, and expansion. We detail them next.

Tokenization. The first step is to split each label into a list of tokens. The key property of the tokenization is that it aims to be vendor-agnostic, i.e., it does not attempt to infer the structure of the labels used by each AV vendor. The reasons for this design decision are that the number of AV vendors is very large (e.g., over

[1] https://github.com/malicialab/avclass.

a hundred AV engines have been used by VirusTotal over time), that each vendor provides different information in their AV labels, and that AV vendors are not very consistent in the format of their labels, often modifying them over time. If we were to try to define, or automatically infer, templates for each label format, we would end up with hundreds of templates. This would make selecting the right template for a label challenging. Worse, whenever a label was observed using a previously unknown label format, tokenization errors could be introduced.

Tagging. The main step in AVclass2 is replacing a token in an AV label with a set of tags in the taxonomy. Such tagging converts the unstructured information in the AV labels into well-defined concepts in the taxonomy. It also identifies potentially useful tokens not yet tagged. Tagging uses a set of input *tagging rules* that specify the set of tags that a token will be replaced with, i.e., a tagging rule maps an input token to an output set of tags. Most tagging rules map a token to a single tag. For example, a tagging rule maps the *downldr* token to the *downloader* tag. In this case, we say that that the token is an *alias* for the tag, since it captures the same concept as the tag. Multiple tokens may be aliases for the same tag. For instance, another tagging rule maps the *dloader* token also to the *downloader* tag. Thus, tokens *downldr* and *dloader* are both *downloader* aliases. Similarly, tokens *finloski* and *fynloski* are aliases for the *darkkomet* family tag. Aliases enable identifying when different vendors use different tokens to capture the same concept. A tagging rule may also define that a token maps to multiple tags if the token captures multiple concepts. For example, the token *ircbot* maps to tags *irc* and *bot*. It is possible to define a tagging rule that maps a token to a tag that is identical to the token. For example, the *downloader* token maps to the *downloader* tag. However, there is no need to define such rules as they are implicitly handled.

A tagging rule should map a token to the most specific tags possible. For example, a tagging rule could map token *addrop* to the *adware* tag, the *grayware* tag, or both of them. In this case, it is best to map it to the *adware* tag because the default taxonomy, detailed in Section 2.2, captures that *adware* is a child of *grayware*. The expansion step below takes care of making explicit the implicit relationships between tags in the taxonomy. Thus, there is no need to assign the *grayware* tag explicitly at this point.

A tagging rule can also define that a token maps to an empty set of tags. We call such tokens *generic* because they do not provide useful information. Generic tokens are discarded during tagging. Example generic tokens are *malicious* and *application*. It is important to highlight that generic tokens in AVclass2 are different from generic tokens in the original AVclass. In AVclass, a generic token provided no information about the family a sample belongs to. In AVclass2, a generic token provides no useful information at all (family or other). Thus, many tokens that were considered generic by AVclass will be assigned tags by AVclass2, e.g., *dldr* was a generic token in AVclass, but produces tag *downloader* in AVclass2. In fact, generic tokens in AVclass are used as seeds to build the default taxonomy for AVclass2, as explained in Section 2.2.

When a token has not been seen before, it will not have a tagging rule defined for it. In this case, the token should not be discarded because it may still provide useful information, e.g., it may capture a newly discovered family. Thus, unknown tokens are passed to the expansion step.

Expansion. The expansion step uses *expansion rules* that define that a tag implies a set of other tags, where the original tag is included in the expanded set. For example, the *worm* class tag can be expanded to include the *selfpropagate* behavior tag, since a worm self-propagates by definition. Expansion rules allow to more accurately capture how many AV engines capture the same concept in their labels. For example, prior to expansion, AVclass2 may output that the *worm* tag is assigned by two engines and *selfpropagate* by another three. After expansion, AVclass2 can output that *selfpropagate* is assigned by five engines. More importantly, expansion rules help increase search coverage. For example, if 95% of samples with the *virut* family tag also have the *virus* class tag, the update module will output an expansion rule from *virut* to *virus*. Using that expansion rule, an extra 5% of *virut* samples will be added the *virus* tag and thus will show up in searches for virus samples.

There exist two types of expansion rules: *intra-category* and *inter-category*. Intra-category rules are implicitly defined by the parent-child relationships in the input taxonomy. They capture that a tag in one category implies other tags in the same category. For instance, the taxonomy may define that the *adware* class tag is a child of the *grayware* class tag. Thus, a sample tagged as *adware* can also be tagged as *grayware*. Inter-category rules capture that a tag from one category implies other tags from a different category. They make explicit otherwise implicit tag relationships, e.g., to add the *filemodify* behavior to the *virus* class tag, as a virus by definition modifies infected files, or to add the *spam* behavior to a bot family known to send spam such as *conficker*. Both inter-category and intra-category expansion rules can be automatically identified by the update module. In addition, an analyst can provide AVclass2 with its own expansion rules based on domain knowledge.

2.2 Taxonomy

AVclass2 takes as input a taxonomy that defines parent-child relationships between the tags used by the tagging rules. AVclass2 ships with a default taxonomy so that it can be used out-of-the-box. The taxonomy supported by AVclass2 is structured as a tree. The first level underneath the root comprises categories. Each category forms its own subtree where nodes are tags and edges capture parent-child relationships between tags. Figure 2 shows a simplified taxonomy with only a handful of tags under each of the four default categories: behavior (BEH), class (CLASS), file properties (FILE), and family (FAM). Some categories such as FAM have no intermediate (i.e., non-leaf) tags, they simply contain a set of tags with no structure. Other categories contain intermediate tags that capture parent-child relationships. For example, the CLASS category contains intermediate tag *grayware* indicating that *adware* and *casino* are classes of grayware. Some intermediate tags only provide structure and are not useful for indexing. These appear capitalized, e.g., *OS*.

By design, AVclass2 assumes that the input taxonomy is incomplete and will not contain all malware-related concepts, not even all concepts that may appear in AV labels. We believe that building a closed malware taxonomy that contains all malware-related concepts is a futile effort, as it is nearly impossible to be complete and

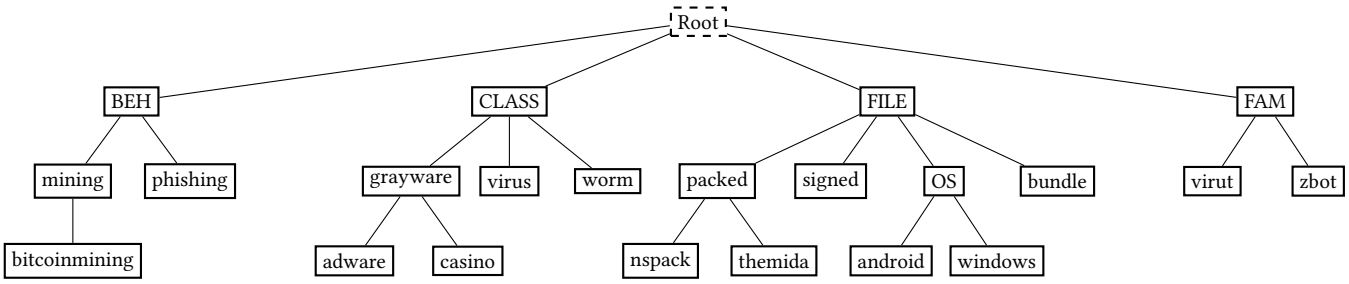

Figure 2: A simple taxonomy with all four categories in the default taxonomy, but only a small subset of its tags.

Table 1: Summary of the default taxonomy.

Category	Tags	Leaf	Intermediate
BEH	44	38	*browsermodify*, *infosteal*, *killproc*, *mining*, *sms*
CLASS	32	23	*adware*, *bot*, *dialer*, *grayware*, *hoax*, *miner*, *tool*, *virus*, *worm*
FAM	894	894	-
FILE	95	88	*exploit*, *FILETYPE*, *installer*, *OS*, *packed*, *PROGLANG*, *signed*

malware continuously evolves, requiring constant updates to any taxonomy. Instead, a key property of AVclass2 is that it supports previously unknown concepts, not yet in the tagging rules and taxonomy. When an *unknown token*, i.e., a token without a tagging rule, appears in an AV label, AVclass2 will keep it and potentially include it in the output, marking it as having an unknown (UNK) pseudo-category. Another key property of AVclass2 is that the taxonomy is provided as a separate input, so that it can be easily modified. Thus, we say that AVclass2 uses an *open taxonomy* that while incomplete, can be refined using the update module or in a collaborative manner.

A tag can be represented by its name or by its path in the taxonomy up to the category it belongs to. For example, the *adware* tag can also be represented as *CLASS:grayware:adware*.

Default taxonomy. AVclass2 provides a default taxonomy, which organizes concepts that commonly appear in AV labels. Since the default taxonomy is included in the open source release of AVclass2, users can share their updates collaboratively. The default taxonomy is summarized in Table 1. It comprises of over 1,000 tags split into the four categories in Figure 2, which we detail next.

- **Behavior (BEH).** Captures how the malware behaves, e.g., *infosteal*, *sendssms*, *spam*, *mining*. Behaviors manifest during a sample's execution. However, once encoded in an AV label, they can be extracted by AVclass2 without the need to execute the sample.
- **Class (CLASS).** Malware classes are widely used to capture malware characteristics such as specific behaviors or distribution methods. Common malware classes are *worm*, *virus*, *ransomware*, and *downloader*. A malware family can belong to multiple malware classes. For example, *wannacry* was both *ransomware* and *worm*. A problematic class is *trojan*. Originally, this class captured a distribution method, namely that the sample fooled the user by claiming fake functionality. However, nowadays *trojan* is the default class used by AV vendors for samples without a more specific class. Thus, we believe it currently holds little meaning and should be considered generic. For this reason, AVclass2 by default considers *trojan* a generic token, but the analyst can easily modify this behavior.
- **File properties (FILE).** Comprises of static file properties including the file type (e.g., *pdf*, *flash*, *msword*), the operating system under which the sample executes (e.g., *android*, *linux*, *windows*), the packer used for obfuscation (e.g., *pecompact*, *themida*, *vmprotect*), and the programming language used to code the sample (e.g., *autoit*, *delphi*, *java*).
- **Family (FAM).** The malware family of the sample. The default taxonomy does not include parent-child relationships between malware families, i.e., no intermediate family tags. Family relationships could be added to the taxonomy using malware lineage approaches [15, 22].

The starting point to build the default taxonomy was the file with generic tokens in the AVclass repository. In AVclass, generics were common tokens in AV labels that do not provide family information. That differs from AVclass2 where generics are tokens that do not provide any useful information, family or other. Our insight was that the AVclass generics contained much useful non-family information such as malware classes, behaviors, and file properties. To build the seed taxonomy, we examined AVclass generics, manually classifying them into the four tag categories above, or as generics that provide no useful information. After that initial manual effort, we refined the seed taxonomy by running the update module on different datasets, incorporating the newly found tags and relationships, as well as adjusting any conflicts it identifies. Table 1 summarizes the current status of the default taxonomy.

2.3 Update Module Overview

Malware is an ever-evolving ecosystem where new concepts keep appearing. Over time, known families exhibit new behaviors and file properties (e.g., novel obfuscations); new families are introduced with their corresponding aliases; and novel malware classes are occasionally created. Furthermore, new relations among known concepts are learned by the community increasing our knowledge base such as which families belong to a certain class or exhibit a specific behavior. Keeping AVclass2 up-to-date with this natural

```
2.VIPRE              "Trojan.Win32.CoinMiner.c"
3.Agnitum            "Bebeg.RiskTool.BitCoinMiner!FwnP7UHRdLc"
4.AVG                "Skodnab.BitCoinMiner.EQ"
```

(a) Input AV labels

```
2.VIPRE              "trojan","win","coinminer","c"
3.Agnitum            "bebeg","risktool","bitcoinminer"
4.AVG                "skodna","bitcoinminer", "eq"
```

(b) After tokenization

```
2.VIPRE              "windows","bitcoinminer"
3.Agnitum            "bebeg","grayware","tool","bitcoinminer"
4.AVG                "skodna","bitcoinminer"
```

(c) After tagging

```
                     "CLASS:miner:bitcoinminer","CLASS:miner",
                     "BEH:mining:bitcoinmining","BEH:mining"
2.VIPRE              "FILE:os:windows",
                     "CLASS:miner:bitcoinminer","CLASS:miner",
                     "BEH:mining:bitcoinmining","BEH:mining"
3.Agnitum            "FAM:bebeg",
                     "CLASS:grayware","CLASS:grayware:tool",
                     "CLASS:miner:bitcoinminer","CLASS:miner",
                     "BEH:mining:bitcoinmining","BEH:mining"
4.AVG                "UNK:skodna",
                     "CLASS:miner:bitcoinminer","CLASS:miner",
                     "BEH:mining:bitcoinmining","BEH:mining"
```

(d) After expansion

```
CLASS:miner:bitcoinminer          3
CLASS:grayware                    2
BEH:mining                        6
BEH:mining:bitcoinmining          3
FAM:bebeg                         2
```

(e) Output

Figure 3: Running example.

evolution is a challenge. It requires to constantly evolve the taxonomy, tagging rules, and expansion rules with new concepts and previously unknown relations. Those new concepts will appear in the output of AVclass2 as unknown tokens, not present in the taxonomy and tagging rules. Manually categorizing those unknown tokens does not scale to the huge numbers of new samples a security vendor may observe each day. We need automatic approaches to keep labeling and tagging tools up-to-date.

To this end, AVclass2 provides an update module to automatically update the input taxonomy, tagging rules, and expansion rules with new concepts and relations. The update module first identifies co-occurrence relations of tokens in AV labels. Co-occurrence in AV labels was introduced in VAMO [32] and later used in AVClass [35] and Euphony [9] to identify family aliases. But, the update module in AVclass2 takes this concept a step further by introducing a novel recursive process that first identifies relations between unknown tokens and tags and then uses a set of learning rules to classify those relations and propose updates to the taxonomy, tagging rules, and expansion rules.

3 LABELING MODULE

The labeling module takes as input the AV labels assigned by multiple AV engines to the same set of samples, an optional list of AV engines whose labels to use, a set of tagging rules, an optional set of expansion rules, and a taxonomy. For each input sample, it outputs a set of tags ranked by the number of AV engines including the tag's concept in their label. AVclass2 comprises of three steps: tokenization, tagging, and expansion. To illustrate them we use the running example in Figure 3. The inputs are the labels assigned by four AV engines to the same sample, shown in Figure 3a.

Tokenization. Takes as input an AV label and outputs the list of tokens the label contains. The tokenization in AVclass2 is almost identical to that of AVclass and we refer the reader to the original paper for details [35]. The only significant difference is that AVclass2 does not filter out short tokens (less than four characters) during tokenization, but rather after tagging. This enables AVclass to extract tags from short tokens that correspond to well-known concepts, e.g., *irc*, *bot*. Figure 3b shows the tokens obtained in our running example.

Tagging. Takes as input the tokens obtained from the tokenization and the input tagging rules. For each input token, if a tagging rule exists for the token, it applies it to obtain a list of tags. If the token is generic, it is removed. If no tagging rule exists for the token, it is kept. It outputs a list of identified tags and any remaining unknown tokens. Figure 3c shows the tagging output. Some tokens have been replaced by their tags, e.g., token *win* by tag *windows* and token *risktool* by tags *grayware* and *tool*. Other tokens have been dropped as generic (e.g., *trojan*) or because they are short (e.g., *eq*).

Expansion. Takes as input the file with expansion rules, the taxonomy, and the tags output by the tagging. For each tag, if an expansion rule exists for it, it applies the rule to obtain a larger list of target tags that replaces the tag. Unknown tokens are not affected by the expansion. The expansion first applies the inter-category expansion rules provided as input to AVclass2. Next, it applies the implicit intra-category expansion rules due to the parent-child tag relationships in the taxonomy. For example, in our running example tag *bitcoinminer* implies tag *CLASS:miner*.

Output. For each tag and unknown token, AVclass2 counts the number of AV labels where it appears. This count can be interpreted as a confidence score. Tags and unknown tokens that appear in the label of at most one AV engine are removed. This filters random unknown tokens that earlier steps may have missed, as the likelihood that those appear in labels from multiple AV engines is low, as well as very low confidence tags.

The output of AVclass2 is the list of tags and unknown tokens along with their confidence score. Figure 3e shows the output of our running example where unknown token *skodna*, as well as tag *FILE:os:windows*, have been removed because they only appeared in one label. AVclass2 also provides a compatibility mode with AVclass to output the most likely family for each sample, which corresponds to the highest ranked family tag or unknown token. i.e., *FAM:bebeg* in our running example.

4 UPDATE MODULE

The update process comprises of two steps. When labeling a dataset, AVclass2 outputs co-occurrence statistics between tags and unknown tokens. The larger the dataset, the higher confidence in the identified co-occurrence statistics. The update module takes as input the co-occurrence statistics, the taxonomy, and the tagging and expansion rules. It performs two substeps: identifying strong relations and converting strong relations into updates to the input files. The process of generating co-occurrence statistics and identifying strong relations is similar to the one used by AVclass to detect alias relations [35]. The novel part of the update module is the recursive process and update rules used to automatically generate updates to the taxonomy, tagging rules, and expansion rules.

Table 2: Update module rules.

Cat(t_i)	Cat(t_j)	Rule
UNK	FAM	tagging(t_i, t_j, FAM)
UNK	CLASS	taxonomy(t_i, FAM:t_i)
UNK	BEH	taxonomy(t_i, FAM:t_i)
UNK	FILE	taxonomy(t_i, path(t_j):t_i)
UNK	UNK	taxonomy(t_i, FAM); taxonomy(t_j, FAM)
FAM	UNK	tagging(t_i, t_j, FAM)
FILE	UNK	tagging(t_i, t_j, prefix(t_i))
FAM	FAM	tagging(t_i, t_j, prefix(t_j))
FAM	FILE	expansion(t_i, t_j)
FAM	BEH	expansion(t_i, t_j)
FAM	CLASS	expansion(t_i, t_j)
CLASS	FILE	expansion(t_i, t_j)
CLASS	BEH	expansion(t_i, t_j)

Co-occurrence statistics. The labeler obtains the co-occurrence statistics after tagging and before expansion. For simplicity, in this section we call tags to both tags and unknown tokens. We call *relation* to each pair of tags that appear in labels for the same sample and its co-occurrence statistics. For each relation, the labeler outputs seven values: the tags t_i, t_j, the number of samples each tag appears in the dataset being labeled $|t_i|$, $|t_j|$, the number of samples where both tags appear $|(t_i, t_j)|$, and the fraction of times that both tags appear in the same samples $rel(t_i, t_j) = \frac{|(t_i,t_j)|}{|t_i|}$ and $rel(t_j, t_i) = \frac{|(t_i,t_j)|}{|t_j|}$. The two tags are sorted so that tag t_i is the one that occurs less often, i.e., $|t_i| \leq |t_j|$, which means that $rel(t_i, t_j) \geq rel(t_j, t_i)$.

Identifying strong relations. Given the set of relations output by the labeler, the update module first filters out weak relations. A relation is *strong* if both tags have been seen in enough number of samples and appear in the same samples frequently enough. The first condition keeps only relations where $min(|t_i|,|t_j|) \geq n$ where n is the minimum number of samples where a tag should have been observed. The second condition keeps only relations where $rel(t_i, t_j) \geq T$. Threshold T controls the minimum joint frequency to determine a relation is strong. For strong relations, we say that t_i *implies* t_j, $t_i \Rightarrow t_j$, but t_j may not imply t_i. For example, if family tag *virut* appears in 1M samples and class tag *virus* in 7M samples, and in every sample *virut* appears *virus* also appears ($rel(virut, virus) = 1.0$), then *virut* $\Rightarrow$ *virus*, but *virus* $\not\Rightarrow$ *virut* as there are 6M instances were *virus* is observed without *virut*, likely corresponding to other virus families ($rel(virus, virut) = 0.14$). If both $rel(t_i, t_j) \geq T$ and $rel(t_j, t_i) \geq T$ we say the tags are *equivalent*, $t_i \Leftrightarrow t_j$. Parameters n and T were empirically selected in AVclass and we use their suggested default values of $n = 20$ and $T = 0.94$.

Weak relations and relations including a OS tag are removed. The latter avoids that an expansion rule is created for each family towards its platform, e.g., *virut* $\Rightarrow$ *windows*, *droidkungfu* $\Rightarrow$ *android*.

Updates to taxonomy and tagging. The update module performs a recursive process where each iteration examines the set of remaining relations. The process starts with the identified strong relations. At each iteration, every remaining relation is checked against a set of rules to identify updates to the current taxonomy, tagging, and expansion rules. Processed relations are removed, the rest are kept. Once all relations are examined, the process runs into a new iteration with the remaining relations. Recursion ends when one iteration does not produce any updates or no relations remain.

Each relation is first checked to see if it is already known. A relation is known if it is already captured in the current taxonomy, tagging rules, or expansion rules. For example, *adware* $\Rightarrow$ *grayware* is implicit in the default taxonomy. This check happens before each relation is processed because the taxonomy, tagging, and expansion rules change as relations are processed and a relation that was not known before may become known once other relations have been processed. Known relations are removed. If not known, and the tags are equivalent, a tagging rule is added from t_i to t_j since t_i is the least common of the two tags. If not an equivalence, the relation is processed according to the rules in the top block of Table 2, which are indexed by the categories of the two tags. UNK is a pseudo-category for unknown tags not in the taxonomy (i.e., unknown tokens). All rules, but the last one, handle relations where at least one tag is unknown. These rules create a tagging rule between the two tags or add the unknown tag to the taxonomy with a path prefix indicated by the rule. For example, the top rule captures that an unknown tag implies a family tag, which creates a tagging rule capturing that the unknown tag t_i is an alias for the family tag t_j. Adding a tagging rule $t_i \Rightarrow t_j$ forces t_i to be removed from the taxonomy (if present) and t_j to be added (if it did not exist). The last rule in the block creates a tagging rule for a relation between family tags indicating that t_i is an alias for t_j and thus t_i does not need to be in the taxonomy.

Updates to expansions. Since expansions happen between two tags in the taxonomy it is more efficient to identify them once all rules for unknown tags have been applied and the taxonomy and tagging rules are stable. Once the recursion ends, all remaining rules are examined using the expansion rules in the bottom block of Table 2. The expansion rules capture properties of a family such as its class, file properties or behaviors, as well as a class having specific behaviors or file properties (e.g., using a exploit).

The five categories (including UNK), create 25 distinct category pairs. However, only 13 pairs are considered in Table 2. There exist 12 category pairs without a rule. Our evaluation will show that those pairs constitute less than 1% of relations and when they happen they indicate some collision that we believe is best resolved manually by an analyst.

Once the process finishes the update module outputs the updated taxonomy, tagging and expansion rules. Most updates will be additional tags in the taxonomy and new tagging and expansion rules, but it is possible that some original taxonomy entries and tagging rules have been modified or removed.

5 EVALUATION

This section evaluates AVclass2. First, Section 5.1 presents the datasets used. Then, Section 5.2 details the tagging results and Section 5.3 demonstrates the update module. Section 5.4 illustrates the benefits of AVclass2. Finally, Section 5.5 compares AVclass2 with prior family tagging tools AVclass and Euphony.

Table 3: Datasets used in evaluation.

Dataset	Plat.	GT	Samples	Bin.	Collection
Superset	Mix	✗	42,533,619	✗	08/2008 - 05/2019
Lever et al. [20]	Win	✗	37,817,328	✗	01/2011 - 08/2015
Andropup	And	✗	3,145,283	✗	05/2019 - 09/2019
Miller et al. [26]	Win	✗	1,079,783	✗	01/2012 - 06/2014
Andrubis [23]	And	✗	422,826	✗	06/2012 - 06/2014
Malsign [19]	Win	✓	142,500	✗	06/2012 - 02/2015
AMD [38]	And	✓	24,553	✓	11/2010 - 03/2016
Malicia [29]	Win	✓	9,908	✓	03/2012 - 02/2013
Drebin [1]	And	✓	5,560	✓	08/2010 - 10/2012
Malheur [34]	Win	✓	3,131	✗	08/2009 - 08/2009
MalGenome [39]	And	✓	1,260	✓	08/2008 - 10/2010

Table 4: Percentage of samples for which a tag could be extracted, as well as percentage of tagged samples with a tag from each category.

	Tagged						
Dataset	All	VT≥4	FILE	CLA	BEH	FAM	UNK
Superset	97%	100%	99%	94%	75%	83%	13%
Lever	98%	100%	99%	94%	78%	83%	13%
Andropup	89%	99%	98%	95%	28%	74%	16%
Miller	95%	99%	100%	91%	82%	82%	20%
Andrubis	100%	100%	100%	89%	69%	98%	4%
Malsign	99%	99%	97%	95%	73%	96%	5%
AMD	100%	100%	100%	99%	31%	98%	14%
Malicia	100%	100%	100%	89%	85%	97%	12%
Drebin	100%	100%	100%	92%	71%	98%	5%
Malheur	100%	100%	100%	93%	83%	66%	7%
MalGen.	100%	100%	100%	96%	82%	98%	2%

5.1 Datasets

We evaluate AVclass2 using the 11 datasets in Table 3. For each dataset, the table shows the target architecture of the samples (Windows, Android, or both), whether the samples are labeled with a family name, the number of samples, whether binaries are available (otherwise we only have their hashes), and the collection period. Datasets come from prior works, save for Superset that is the union of all other datasets. Some datasets overlap, e.g., Drebin [1] is a superset of MalGenome [39]. We do not remove duplicates to make it easy to map results to publicly available datasets.

We had VT reports for most samples and only collect a few missing ones. We use the available reports since VT's rate limits make it infeasible for us to collect the latest reports.

5.2 Tagging Evaluation

Table 4 summarizes the tagging coverage of AVclass2, i.e., the fraction of samples for which it can extract a tag. These results are obtained using the tagging rules, expansion rules, and taxonomy output by the update module after it identifies previously unknown relations. In Section 5.3 we analyze the tagging improvement before and after applying the update module. The table first shows the fraction of all samples, and those flagged by at least four AV engines, for which at least one tag was obtained. We use the threshold of four AV detections to remove potentially benign samples. This threshold has been used in prior works [19] and a recent work shows that threshold values between two and 14 are good for stability and for balancing precision and recall [40]. For each category, the table then shows the fraction of tagged samples with at least four detections with a tag of that category. Column UNK corresponds to samples with at least one output unknown token not in the taxonomy.

The results show that AVclass2 can extract at least one tag for 89%–100% of the samples, depending on the dataset. Thus, it is possible to index the majority of samples. The files for which no tags can be extracted largely correspond to those with very few detections, which is higher for Andropup as it is the most recent dataset. When considering sample flagged by at least four AV engines the fraction of tagged samples is at least 99%. As shown by the Superset, the most common tags are file properties (99% of samples), followed by malware classes (94%), known families (83%), and behaviors (75%). It is important to note that the FAM column considers only samples tagged with families that appear in the taxonomy. However, unknown tags most often correspond to new families that have not yet been added to the taxonomy and should be considered for final family tagging results. Section 5.5 compares the family tagging results of AVclass2 to prior tools like AVclass and Euphony.

Most popular tags. Table 5 shows the top 10 tags in the Superset for each category, ranked by the number of samples assigned the tag. The most popular tag is *FILE:OS:windows* (61% of samples), followed by *CLASS:grayware* (46%), *BEH:execdownload* (27%), *BEH:filemodify* (21%), *FILE:packed* (20%), and *CLASS:downloader* (20%). Our taxonomy contains 32–95 tags in each of the non-family categories, and 894 families. Of the tags in the taxonomy 94%–100%, depending on the category, appear in the Superset. Thus, the distribution of tags per category contains a large number of tags with at most 1% coverage, each identifying up to tens of thousands of samples. Thus, AVclass2 extracts a wide variety of tags that can be used by the analyst to search for samples according to class, family, file properties, and behaviors.

The top FILE tags include the platform (*windows*, *android*), whether the sample is *packed*, if it is a *bundle* that contains other executables, the programming language (*autoit*, *msil* for C#), if it is an *installer* or has been generated by a particular installer (*nsis*), and the size of the file (*small*). Intermediate tags rank high because they accumulate popularity of their children through the expansion on taxonomy relationships. For example, *packed* accumulates the influence of all packer tags in the taxonomy (e.g., *asroot*, *upack*, *themida*). And, *installer* the influence of *nsis*, as well as other installer-generating software (e.g., *wiseinstaller*, *installmate*).

There are four CLASS tags appearing in more than 10% of the samples. They capture the popularity of potentially unwanted programs, downloaders, monetizing through advertisements, and viruses. *CLASS:grayware:adware:multiplug* are *adware* that install browser plugins (e.g., extensions, toolbars) to modify the user's Web surfing. And, *CLASS:grayware:tool* are tools not necessarily malicious, but often abused, such as those used for remote administration. Note that *trojan* is considered generic, otherwise it would be assigned to 86% of the samples.

The top three behavior tags correspond to expansions from class tags: *downloader* ⇒ *execdownload*, *virus* ⇒ *filemodify*, *worm*

Table 5: Top 10 tags for each category ranked by the number of samples assigned the tag in the Superset dataset.

Rank	FILE		CLASS		BEH		FAM		UNK	
1	windows	61%	grayware	46%	execdownload	27%	vobfus	10%	fraudpack	1%
2	packed	20%	downloader	20%	filemodify	21%	loadmoney	5%	atraps	1%
3	android	5%	adware	18%	selfpropagate	12%	virut	5%	hiddapp	1%
4	bundle	3%	virus	15%	autorun	10%	softpulse	4%	hiddenads	1%
5	installer	2%	worm	9%	inject	6%	installerex	3%	packer	1%
6	small	2%	backdoor	4%	server	6%	domaiq	3%	llac	1%
7	nsis	1%	multiplug	4%	alertuser	2%	firseria	3%	trymedia	1%
8	msil	1%	rogueware	2%	killproc	2%	zbot	3%	refroso	1%
9	autoit	1%	tool	1%	sms	2%	sality	3%	bifrost	1%
10	html	1%	clicker	1%	ddos	2%	virlock	2%	comame	1%

⇒ *selfpropagate*. Note that the behavior associated to a class gets a higher tagging ratio than its class, indicating additional samples without the corresponding class tag. For example, *execdownload* has an additional 7% samples without the *downloader* class tag and *filemodify* an additional 6% over *virus*. Other popular behaviors are *autorun*, which captures samples that modify the `autorun.inf` Windows file to automatically execute; those related to obfuscation such as injecting code in a benign process and killing a process, typically of a security tool; opening a server, alerting the user, sending SMS messages; and launching denial of service attacks.

Top families have lower prevalence than top classes, file properties, and behaviors. The most prevalent family is *vobfus* (10%). Out of the top families, half correspond to *grayware* families (*loadmoney*, *softpulse*, *installererex*, *domaiq*, *firseria*) and the rest are malware. Top unknown tokens have much lower prevalence, at most 1%. They mostly correspond to families not yet in the taxonomy, for which no strong relation has yet been observed to another tag.

5.3 Update Module

We illustrate the usage of the update module with the Andropup dataset. When we first obtained this dataset, we run AVclass2's labeler observing that 65% of samples contained an unknown tag. To reduce this number we run the update module on the current taxonomy and without any expansion rules. The updated taxonomy, tagging rules, and expansion rules reduced the samples with an unknown tag from 65% to the 16% shown in Table 4.

Table 6 summarizes the update module results. Since we only have the same family ground truth used in AVclass, we use the default values suggested in that work of $n = 20$ and $T = 0.94$ [35]. The co-occurrence statistics output by the labeler contain 30,107 relations, of which 968 are strong. Those 968 relations belong to 11 category pairs, the most popular being UNK ⇒ CLASS (61%), UNK ⇒ UNK (17%), UNK ⇒ BEH (9%), UNK ⇒ FAM (7%), FAM ⇒ CLASS (2%), and UNK ⇒ FILE (2%). The other five category pairs had at most 0.3% relations each. Overall, 96% of the strong relations involve an unknown token.

From those 968 strong relations, the update module automatically identified 486 new taxonomy entries, 216 new tagging rules, and 461 expansion rules. Of the new taxonomy entries, 97% correspond to new families. The other 3% are file properties, mostly Android exploits, e.g., *FILE:exploit:asroot*, *FILE:exploit:exploid*, and *FILE:exploit:gingerbreak*. The new expansion rules link families to their class or behavior. The most popular destination tags are: *grayware* (46%), *adware* (26%), *infosteal* (8%), and *downloader* (6%).

At the end of the update module processing, there remain only three strong relations (0.3%) for which no updates are generated because they lack a processing rule. For these relations the analyst would have to manually decide how to handle them. The first is *FILE:patch* ⇒ *CLASS:grayware*. This is an example of an *homonym*, i.e., a token with two possible meanings. In our seed taxonomy, obtained from the AVclass generics, we manually (incorrectly) classified *patch* as a file type. But, this relation automatically extracted by the update module allowed us to understand that the tag is used for modding apps (e.g., Lucky Patcher[2]) that patch other apps for unlimited access, no ads, etc. This allowed us to correct our error making *patch* an alias for *BEH:filemodify*. Another homonym we incorrectly classified in our seed taxonomy is *fakedoc*, which we though meant a fake document, but the update module identified that *batterydoctor* ⇒ *FILE:fakedoc*, indicating *fakedoc* is instead a rogueware family (*fakedoctor*). The other two unhandled relations are due to over-fitting of a particular family. Relation *FILE:proglang:powershell* ⇒ *CLASS:keylogger* is due to a keylogging family that uses Powershell scripts and relation *BEH:inject* ⇒ *CLASS:downloader* to a downloader family that injects in other processes. Since they are not general enough, we ignore both relations.

Update quality. To evaluate the quality of the generated updates, we manually examine the resulting taxonomy, tagging, and expansion rules looking for errors based on our domain knowledge and Web searches. We acknowledge that this process is subjective and may not spot all errors, but it is the best ground truth we were able to obtain. We identify 11 cases were we would have done things differently than the update module. First, five family tags should likely be classes/subclasses (*bankbot*, *clickfraud*, *fakeantivirus*, *locker*, *remoteadmin*). In addition, *trojandldr* was marked as a family tag with an expansion rule *trojandldr* ⇒ *downloader*, but making the expansion rule a tagging rule would have been cleaner. One challenge is distinguishing the name of an exploit (file property) from a family that uses an exploit. In most cases the update module is correct, but *FILE:exploit:rootmaster* should likely be *FAM:rootmaster* with an expansion rule *rootmaster* ⇒ *exploit*. There is also one tagging rule *cryptominer* ⇒ *coinhive* where *cryptominer* should likely be an alias for *CLASS:miner*. One new file property in the taxonomy

[2] https://www.luckypatchers.com/

Table 6: Update module evaluation results with $n = 20$ and $T = 0.94$.

	Relations			Taxonomy Entries		Tagging Rules		Expansion Rules	
Dataset	**All**	**Strong**	**Out**	**Added**	**Rem.**	**Added**	**Rem.**	**Added**	**Rem.**
Andropup	30,107	968	3	486	2	216	10	461	0

FILE:proglang:java:genericgba is likely a generic token. And, another file property *FILE:packed:decrypter* does not look like any known packer, so we are not sure what it exactly represents. Finally, *FILE:testvirus:testfile* would be better as a tagging rule *testfile* ⇒ *testvirus*.

In summary, out of 1,163 updates to the taxonomy and rules, only 11 (0.9%) required adjustments. Additionally, three relations had to be checked, allowing us to correct errors with homonyms introduced in the manual creation of the seed taxonomy.

5.4 Beyond Family Labeling

The main benefits of AVclass2 over prior family labeling tools are that it enables tag-based searches beyond families, and it builds malware knowledge generalizing beyond individual samples.

Search. The tags output by AVclass2 enable advanced searches on a malware repository. The 975 tags identified in the Superset, can be used, among others, to identify samples that are grayware, belong to a specific class (e.g., ransomware, information stealers), send SMS, launch DoS attacks, use a specific packer (e.g., *themida*, *asroot*), or installer software (e.g., *nsis*, *wiseinstaller*) software. Those samples can then be used to build classifiers.

We examine how the extracted tags compare with those already used by VirusTotal. The VT documentation mentions 335 tags, mostly corresponding to file properties and behaviors [37]. Those 335 tags do not include family names or classes (save for *worm* and *emailworm*). Of those 335 tags, we observe 259 in the reports of samples in the Superset. Of those VT tags, 49 are identical to tags in our default taxonomy with the largest types being 30 packers, 4 programming languages, and 4 file types. Thus, the tags output by AVclass2 nicely complement the ones already in use by VT. Since AVclass2 already operates on VT reports, adding AVclass2's tags to VT would be straightforward and would enable new searches not currently possible such as those for samples of a particular malware class or family, as well as searches for a richer set of file properties and behaviors. This highlights the benefit of AVclass2's automatic tag extraction to popular malware repositories.

Knowledge base. AVclass2 taxonomy, tagging rules, and expansion rules form a malware knowledge base that capture relations beyond individual samples such as which families are ransomware or information stealers. Compared to existing online malware encyclopedias, its contents have been obtained using a well-defined methodology; are not specific to one vendor; and since they are publicly available, can be discussed and evolved collaboratively. Furthermore, AVclass2's update module allows to refine the knowledge base over time keeping it up-to-date.

5.5 Family Labeling Comparison

This section compares AVclass2 with AVclass and Euphony for family labeling. The goal is not to evaluate AVclass2 for family labeling, as an analyst that only requires family labeling, but not the additional malware and threat intelligence AVclass2 provides, does not need to use AVclass2, as it does not significantly differ from AVclass for that task. Instead, the comparison allow us to (1) evaluate AVclass versus Euphony (not done in the original works), (2) highlight the importance of the update module, and (3) show our vision for using AVclass2 and AVclass.

For the comparison, we use the most recent versions of Euphony and AVclass at the time of writing. and the Windows and Android datasets with ground truth. While Euphony was only evaluated on labels of Android samples, its processing is equally valid for Windows. For AVclass2, we use its AVclass compatibility option, which selects as the sample's family the highest ranked family tag or unknown token. For AVclass, we use the most recent aliases and generics in its repository. We also include in the comparison AVclass*, a modified version of AVclass that takes as input the taxonomy and tagging rules from AVclass2, instead of the alias and generic files in the AVclass repository. AVclass* extracts from the taxonomy and tagging rules the list of aliases and generic tokens AVclass needs. The reason to include AVclass* is to compare AVclass2 and AVclass on the same input knowledge base, ignoring any differences due to the knowledge base of AVclass2 being more up-to-date.

Results of the four tools, run on the same server, are in Table 7. For each tool and dataset it shows the runtime in seconds, the percentage of samples with a family, and the accuracy (precision, recall, F1 score) when the results are compared with the available ground truth. Given that these are old datasets with large average number of detections per sample, all tools output a label for almost all samples. AVclass2 and AVclass* achieve the highest F1 score on four of the five datasets, while AVclass ranks first on Malheur. Euphony's accuracy drops significantly on the larger datasets from 89%-91% on Malheur and Drebin down to 59%-73% on Malsign, Malicia, and AMD. AVclass2 and AVclass do not show such drop. AVclass is the fastest, followed by AVclass2, with Euphony being from 7 to 34 times slower than AVclass. Furthermore, we tested Euphony on the larger datasets without ground truth and for those larger than 1M samples, it did not terminate in 48 hours or crashed with an out of memory exception (on a server with 126 GB RAM).

In summary, AVclass is more accurate and 7x–34x faster than Euphony. Furthermore, Euphony does not scale to large datasets due to its large memory usage. AVclass* and AVclass2 have the same accuracy, but AVclass2 is slower due to the extra processing to extract more tags. Thus, there is no reason to use AVclass2 over AVclass for family labeling, as long as both tools use the same input knowledge. Without providing updated data files to AVclass, AVclass2 would outperform it, despite the family labeling functionality being identical. This highlights the importance of the update module.

Table 7: Family labeling comparison between AVclass2, AVclass, and Euphony on datasets with ground truth. T is in seconds.

	AVclass2					AVclass *					AVclass					Euphony				
Dataset	**T**	**Fam**	**Prec**	**Rec**	**F1**	**T**	**Fam**	**Prec**	**Rec**	**F1**	**T**	**Fam**	**Prec**	**Rec**	**F1**	**T**	**Fam**	**Prec**	**Rec**	**F1**
Malsign	205	99%	91%	92%	92%	139	99%	91%	92%	92%	139	100%	90%	90%	90%	2,126	97%	74%	49%	59%
AMD	44	100%	92%	80%	86%	50	100%	92%	80%	86%	50	100%	92%	76%	83%	331	100%	76%	70%	73%
Malicia	20	100%	95%	60%	74%	14	100%	95%	60%	74%	14	100%	95%	61%	74%	593	100%	86%	54%	67%
Malheur	2	100%	91%	94%	92%	1	100%	91%	94%	92%	1	100%	91%	98%	94%	53	100%	90%	93%	91%
Drebin	5	100%	94%	96%	95%	8	100%	94%	96%	95%	8	100%	96%	89%	92%	208	100%	94%	85%	89%

An analyst that requires advanced searches or the extra intelligence should use AVclass2, but those that exclusively need family labels could still use the faster AVclass. Our vision is that in the future both AVclass2 and AVclass take as input the same data files, avoiding duplicate work to keep both tools up-to-date. For this, we plan to integrate our AVclass* modifications into AVclass, so that both tools take as input the same files. This will enable the community to collaboratively update the malware knowledge in the taxonomy, tagging rules, and expansion rules, avoiding duplicate efforts, while maintaining backwards compatibility.

6 RELATED WORK

AV labels have been widely studied for over a decade. Early works showed the problem of different AV engines disagreeing on labels for the same sample [2, 5]. Despite this problem, AV labels have been widely used to build training datasets and evaluate malware detection and clustering approaches [1–3, 7, 12, 14, 19, 25, 26, 30, 31, 33, 34]. Li et al. [21] studied AV labels as a reference to evaluate clustering results. They showed that including only a subset of samples, for which AV engines largely agree, biases the evaluation towards samples that are easier to label. Other works have focused on the quality of the labels from different AV engines. Mohaisen and Alrawi [28] proposed metrics for evaluating AV labels, identifying clusters of AV engines that copy their labels [28]; Kantchelian et al. [17] discussed that AV engines have varying label quality and proposed to weight them differently; and Hurier et al. [13] proposed metrics to evaluate ground truth datasets built using AV labels.

The dynamics of AV labels have been another target of analysis. Some works have shown how AV engines change their labels for the same samples over time as a result of signatures and analysis being refined [10, 17] and that detection systems should be trained with the labels available at training, not testing, time [26]. Recently, Zhu et al. [40] analyze daily snapshots of 14K samples over a year. Among other results, they confirm that certain sets of AV engines produce strongly correlated labels, as observed in AVMeter [28] and implemented in AVClass [35], show that hand-picking of a few trusted engines does not always perform well, and measure that detection thresholds between two and 14 exhibit little differences in precision and recall.

Malware labeling. One approach to tackle disagreements on malware names is to use naming conventions such as the 1991 CARO Virus Naming Convention [6]. Another attempt was the Common Malware Enumeration (CME) Initiative [4] that provided unique identifiers for malware. Unfortunately, conventions have not achieved wide adoption, possibly due to their use of predefined tags, i.e., controlled vocabularies, that are incomplete and require frequent updates.

An alternative approach is to automatically extract accurate family names from AV labels. A precursor of this was VAMO [32] that proposed an automated approach for evaluating clustering results by building a graph of the normalized labels of 4 AV engines. It introduced the use of label co-occurrence, i.e., the fraction of samples where labels, possibly from different engines, appear together.

Tools like AVclass [35] and Euphony [9] have demonstrated that it is possible to extract accurate family tags from AV labels. One key idea of these works is to avoid using the whole label as family name, as AV labels encode other non-family information. AVclass and Euphony take as input AV labels for a large number of samples, and output a family name for each sample. They both use co-occurrence to automatically identify family aliases. However, there are some key differences between them. AVclass proposes that aliases and generic tokens learned from one dataset can be reused on other datasets. Instead, Euphony uses a graph-based approach that identifies aliases within the dataset, but does not produce relations that can be reused. In addition, AVclass avoids predicting which AV engines are better at labeling samples. Instead, it makes the tokenization as AV engine independent as possible. Instead, Euphony tries to learn the structure of the labels from a selected subset of AV engines. As shown in Section 5.5, these differences result in improved accuracy and scalability for AVclass, a key reason why we build AVclass2 on top of AVclass.

Compared to prior malware labeling tools, AVclass2 leverages that AV labels contain a wealth of information beyond family names such as malware classes, file properties, and behaviors. Such information is not extracted by tools like AVclass and Euphony and can be used to produce tags to categorize and index malware samples.

Malware tagging. The use of tags to characterize malware is at the heart of information sharing standards like Malware Attribute Enumeration and Characterization (MAEC) [24] and Malware Information Sharing Platform (MISP) [27]. In addition, tags are already used by some malware repositories to enable efficient search. To further the use of malware tags, we present AVclass2, an automatic tool to extract tags from the wealth of information on AV labels. The generated tags can be incorporated by those repositories to enable richer searches. Most related to our work is simultaneous work by Ducau et al. [8] that extract 11 pre-defined tags, mostly malware classes, from the AV labels of 10 AV engines. They use the tags as training set for a classifier that predicts the tags in new samples. Some key differences of AVclass2 are that it does not pre-define the tags, instead building an open taxonomy (which currently has 150 non-family tags), handles tag aliasing (e.g., *downloader* and *dropper*

are aliases in our taxonomy), provides support for updating the input rules and taxonomy, does not limit the supported AV engines, and is open source.

7 CONCLUSION

Automatically extracting tags from AV labels is an efficient approach to categorize and index massive amounts of malware samples. But, it is challenging due to the different vocabularies used by AV engines. In this work, we have presented AVclass2, an open source automatic malware tagging tool. Given the AV labels for a potentially massive number of samples, AVclass2 extracts clean tags that categorize the samples according to their class, family, file properties, and behaviors. The extracted tags can be used by malware repositories and analysis services to enable, or enhance, searches for samples of interest. Those samples, can in turn be used as ground truth for machine learning approaches. AVclass2 uses, and helps building, an *open taxonomy* that organizes concepts in AV labels, but is not constrained to a predefined set of tags. AVclass2 provides an update module that uses tag co-occurrence to automatically identify taxonomy updates, as well as tagging and expansion rules that capture relations between tags. Thus, it can be easily updated as AV vendors introduce new tags. We have evaluated AVclass2 on 42M samples.

ACKNOWLEDGMENTS

This research was supported by the Regional Government of Madrid through grants BLOQUES-CM P2018/TCS-4339 and PEJD-2018-PRE/TIC-9571 and by the Spanish Government through the SCUM grant RTI2018-102043-B-I00 and fellowship FPU18/06416. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors or originators, and do not necessarily reflect the views of the sponsors.

REFERENCES

[1] Daniel Arp, Michael Spreitzenbarth, Malte Huebner, Hugo Gascon, and Konrad Rieck. 2014. Drebin: Efficient and Explainable Detection of Android Malware in Your Pocket. In *Network and Distributed System Security*.

[2] Michael Bailey, Jon Oberheide, Jon Andersen, Zhuoqing Morley Mao, Farnam Jahanian, and Jose Nazario. 2007. Automated Classification and Analysis of Internet Malware. In *International Symposium on Recent Advances in Intrusion Detection*.

[3] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. 2009. Scalable, Behavior-Based Malware Clustering. In *Network and Distributed System Security*.

[4] Desiree Beck and Julie Connolly. 2006. The Common Malware Enumeration Initiative. In *Virus Bulletin Conference*.

[5] Julio Canto, Marc Dacier, Engin Kirda, and Corrado Leita. 2008. Large Scale Malware Collection: Lessons Learned. In *IEEE SRDS Workshop on Sharing Field Data and Experiment Measurements on Resilience of Distributed Computing Systems*.

[6] CARO [n.d.]. CARO Virus Naming Convention. http://www.caro.org/articles/naming.html.

[7] George E. Dahl, Jack W. Stokes, Li Deng, and Dong Yu. 2013. Large-Scale Malware Classification using Random Projections and Neural Networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing*.

[8] Felipe N Ducau, Ethan M Rudd, Tad M Heppner, Alex Long, and Konstantin Berlin. 2019. SMART: Semantic Malware Attribute Relevance Tagging. *arXiv preprint arXiv:1905.06262* (2019).

[9] Euphony [n.d.]. Harmonious Unification of Cacophonous Anti-Virus Vendor Labels for Android Malware. https://github.com/fmind/euphony.

[10] Ilir Gashi, Bertrand Sobesto, Stephen Mason, Vladimir Stankovic, and Michel Cukier. 2013. A Study of the Relationship between Antivirus Regressions and Label Changes. In *International Symposium on Software Reliability Engineering*.

[11] Harry Halpin, Valentin Robu, and Hana Shepherd. 2007. The Complex Dynamics of Collaborative Tagging. In *International Conference on World Wide Web*.

[12] Wenyi Huang and Jack W. Stokes. 2016. MtNet: A Multi-Task Neural Network for Dynamic Malware Classification. In *Detection of Intrusions and Malware, and Vulnerability Assessment*.

[13] Médéric Hurier, Kevin Allix, Tegawendé Bissyandé, Jacques Klein, and Yves Le Traon. 2016. On the Lack of Consensus in Anti-Virus Decisions: Metrics and Insights on Building Ground Truths of Android Malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment*.

[14] Jiyong Jang, David Brumley, and Shobha Venkataraman. 2011. BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *ACM Conference on Computer and Communications Security*.

[15] Jiyong Jang, Maverick Woo, and David Brumley. 2013. Towards Automatic Software Lineage Inference. In *USENIX Security Symposium*.

[16] JoeSandbox [n.d.]. Joe Sandbox. https://www.joesandbox.com/.

[17] Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Brad Miller, Vaishaal Shankar, Rekha Bachwani, Anthony D Joseph, and JD Tygar. 2015. Better Malware Ground Truth: Techniques for Weighting Anti-Virus Vendor Labels. In *ACM Workshop on Artificial Intelligence and Security*.

[18] Christian Körner, Dominik Benz, Andreas Hotho, Markus Strohmaier, and Gerd Stumme. 2010. Stop Thinking, Start Tagging: Tag Semantics Emerge from Collaborative Verbosity. In *International Conference on World Wide Web*.

[19] Platon Kotzias, Srdjan Matic, Richard Rivera, and Juan Caballero. 2015. Certified PUP: Abuse in Authenticode Code Signing. In *ACM Conference on Computer and Communication Security*.

[20] Chaz Lever, Platon Kotzias, Davide Balzarotti, Juan Caballero, and Manos Antonakakis. 2017. A Lustrum of Malware Network Communication: Evolution and Insights. In *Proceedings of the 38th IEEE Symposium on Security and Privacy*. San Jose, CA, USA.

[21] Peng Li, Limin Liu, Debin Gao, and Michael K Reiter. 2010. On Challenges in Evaluating Malware Clustering. In *International Symposium on Recent Advances in Intrusion Detection*.

[22] Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. 2012. Lines of Malicious Code: Insights into the Malicious Software Industry. In *Annual Computer Security Applications Conference*.

[23] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. 2014. ANDRUBIS-1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*.

[24] MAEC [n.d.]. Malware Attribute Enumeration and Characterization. http://maec.mitre.org/.

[25] Federico Maggi, Andrea Bellini, Guido Salvaneschi, and Stefano Zanero. 2011. Finding Non-Trivial Malware Naming Inconsistencies. In *International Conference on Information Systems Security*.

[26] Brad Miller, Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Rekha Bachwani, Riyaz Faizullabhoy, Ling Huang, Vaishaal Shankar, Tony Wu, George Yiu, Anthony D. Joseph, and J. D. Tygar. 2016. Reviewer Integration and Performance Measurement for Malware Detection. In *Detection of Intrusions and Malware, and Vulnerability Assessment*.

[27] misp [n.d.]. MISP Standard. https://www.misp-standard.org/.

[28] Aziz Mohaisen and Omar Alrawi. 2014. AV-Meter: An Evaluation of Antivirus Scans and Labels. In *Detection of Intrusions and Malware, and Vulnerability Assessment*.

[29] Antonio Nappa, M. Zubair Rafique, and Juan Caballero. 2015. The MALICIA Dataset: Identification and Analysis of Drive-by Download Operations. *International Journal of Information Security* 14, 1 (February 2015), 15–33.

[30] Roberto Perdisci, Andrea Lanzi, and Wenke Lee. 2008. McBoost: Boosting Scalability in Malware Collection and Analysis using Statistical Classification of Executables. In *Annual Computer Security Applications Conference*.

[31] Roberto Perdisci, Wenke Lee, and Nick Feamster. 2010. Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces. In *USENIX Symposium on Networked Systems Design and Implementation*.

[32] Roberto Perdisci and U. ManChon. 2012. VAMO: Towards a Fully Automated Malware Clustering Validity Analysis. In *Annual Computer Security Applications Conference*.

[33] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. 2008. Learning and Classification of Malware Behavior. In *Detection of Intrusions and Malware, and Vulnerability Assessment*.

[34] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. 2011. Automatic Analysis of Malware Behavior using Machine Learning. *Journal of Computer Security* 19, 4 (2011).

[35] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. 2016. AVClass: A Tool for Massive Malware Labeling. In *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions and Defenses*. Evry, France.

[36] VirusTotal [n.d.]. VirusTotal. https://virustotal.com/.

[37] vtTags [n.d.]. Full list of VirusTotal Intelligence tag modifier. https://support.virustotal.com/hc/en-us/articles/360002160378-Full-list-of-VirusTotal-Intelligence-tag-modifier.

[38] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. 2017. Deep Ground Truth Analysis of Current Android Malware. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment*.

[39] Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android Malware: Characterization and Evolution. In *IEEE Symposium on Security and Privacy*.

[40] Shuofei Zhu, Jianjun Shi, Limin Yang, Boqin Qin, Ziyi Zhang, Linhai Song, and Gang Wang. 2020. Measuring and Modeling the Label Dynamics of Online Anti-Malware Engines. (2020).