Euphony: Harmonious Unification of Cacophonous Anti-Virus Vendor Labels for Android Malware

Médéric Hurier*, Guillermo Suarez-Tangil‡, Santanu Kumar Dash†,
Tegawendé F. Bissyandé*, Yves Le Traon*, Jacques Klein*, Lorenzo Cavallaro‡
*University of Luxembourg, Luxembourg. Email: {firstname.lastname@uni.lu}
†University College London, United Kingdom. Email: {firstname.lastname@ucl.ac.uk}
‡Royal Holloway, University of London, United Kingdom. Email: {firstname.lastname@rhul.ac.uk}

Abstract—**Android malware is now pervasive and evolving rapidly. Thousands of malware samples are discovered every day with new models of attacks. The growth of these threats has come hand in hand with the proliferation of collective repositories sharing the latest specimens. Having access to a large number of samples opens new research directions aiming at efficiently vetting apps. However, automatically inferring a reference dataset from those repositories is not straightforward and can inadvertently lead to unforeseen misconceptions. On the one hand, samples are often *mis-labeled* as different parties use distinct naming schemes for the same sample. On the other hand, samples are frequently *mis-classified* due to conceptual errors made during labeling processes. In this paper, we mine Anti-Virus labels and analyze the associations between all labels given by different vendors to systematically unify common samples into family groups. The key novelty of our approach, named EU-PHONY [20], is that no a-priori knowledge on malware families is needed. We evaluate EUPHONY using reference datasets and more than 400 thousands additional samples outside of these datasets. Results show that EUPHONY can accurately label malware with a fine-grained clustering of families, while providing competitive performance against the state-of-the-art.**

Keywords-**malware; android; ground-truth; datasets; labeling;**

I. INTRODUCTION

The popularity of the Android platform with consumers has made it a prime target of malware developers. In recent years, new and evolved models of attacks are regularly developed and thousands of malicious samples are discovered every month. For classifying malware in the wild, which is central to mitigation techniques, researchers have proposed approaches for automatically classifying malware using both supervised machine learning [4], [15], [16], [28] and clustering [41], [44]. A crucial element in all such approaches to build learning models for malware classification are reference datasets for training and evaluating the model. Unfortunately, as Rossow et al. [35] have pointed out, the literature exhibits several shortcomings, including a lack a correctness, transparency and realism in the handling of malware datasets. In particular, reliable malware labels are a necessary input to guarantee the quality of both malware detection and classification models.

Malware labeling, however, is not a trivial task. Manual labeling, where a human analyst inspects the actions of the malware in a bid to classify them, is prohibitively expensive, given the number of malware samples discovered every day. In such a setting, it is reasonable to rely on the collective judgment of Anti-Virus (AV) vendors who specialize in malware labeling. However, deriving a unified label from labels attached to samples by AV vendors is difficult. Inconsistencies in Anti-Virus (AV) labels are indeed common. This is due to both naming disagreements [24], [25] across vendors and also a lack of adopted standards[1] for naming malware.

Prior work has relied on simple heuristics to come up with unified labels based on assessment reports of AV vendors. For the case of labeling malware as benign or malicious, techniques have labeled a sample as malicious if at least one AV vendor flags it as malicious or at least majority of AV vendors have flagged it as malicious [4], [28]. While such heuristics work for flagging samples as malicious or benign, labeling samples with the specific class they belong to is fraught with difficulties. AV vendors can choose different norms to name classes, prefixing qualifying attributes such as attack type (e.g., *Trojan*) or platform (e.g., *Android*) to the label. What further complicates things is that it is not uncommon for typographic and orthographic inconsistencies to creep into the labeling process not just across vendors but sometimes even for the same vendor. Consequently, a sample's full AV label is a poor indicator of its generic family name. For example, the family name *Adrd* is "lost" in the full AV label *Android.Trojan.Adrd.A (B)*.

In this work, we present EUPHONY [20], a tunable AV labelling system that can systematically extracts information from AV labels and learns their patterns and vocabulary over time. EUPHONY is an inference-based system, which allows for end-to-end automation, relieving practitioners from the need to collect, aggregate and verify malware families manually. EUPHONY's label-unification scheme is also vendor agnostic. No specific rules about AV engines (e.g., which label parts are suffixes to be removed) are encoded in the process as these rules are inferred from the available AV labels data.

We report in this paper on the following contributions:

- We present the design and implementation of EUPHONY which aims at clustering malware samples in the wild and inferring their family names. Evaluated on Genome [47] and *Drebin* [4] reference datasets of Android malware, EUPHONY achieves relatively high F-measure performance of 92.7% and 95.5% respectively.

---

[1]CARO and CME conventions are not widely used by AV vendors.

- EUPHONY, in contrast with previous work, is able to automatically learn the structure and lexicon of AV labels to iteratively improve its performance on family inference over time.
- Finally, we provide to the community a new, and larger, reference dataset of Android malware with labels inferred by EUPHONY. We also interface EUPHONY with the public API of the *Androzoo* project [3], [42] to regularly provide the community with up-to-date reference datasets, allowing for reliable and reproducible experiments within the community.

II. RELATED WORK

Android Malware and Machine Learning: Research on Android malware has attracted a lot of attention from the security community due to the popularity of the platform and the continuous rise of threats in this ecosystem [18], [19], [21], [23], [33], [34], [46], [48]. In an effort to automatically detect and classify malicious apps, several machine learning approaches have been recently proposed [1], [4], [8], [11], [14], [22], [30], [36], [45]. These solutions have shown promising results in closed environments, but they also require a large dataset of malware samples in order train the system efficiently.

While such sets are readily available in other domains, they are not as easy to obtain for security purposes. If we take the case of Android malware, the main reference dataset, *MalGenome* [47] and *Drebin* [4], are now more than four years old. Meanwhile, practitioners have collected new apps from online markets [3] but cannot guarantee that an app is indeed malicious, or to which family it belongs, without proper vetting mechanisms. This is a chicken and egg problem which impers the use of machine learning techniques in the outside world [38].

Our work addresses this challenge by providing a tool to associate meaningful families to malware samples and support these automated analysis techniques.

The Reference Dataset Problem: Reference datasets are essential to devise new detection patterns and systematically analyze unknown malware samples. However, despite their importance, they are rarely fully qualified by the research community. In an effort to shed light on this issue, Rossow et al. [35] have studied the methodological rigour of 36 academic publications that rely on malware execution. They pointed out several shortcomings, including a lack a correctness, transparency and realism in the handling of malware datasets. Moreover, Sommer and Paxson [38] have raised similar concerns in an attempt to explain the gap between closed world and real world experiments.

Several reasons can be put forward to explain the notorious difficulty of composing such reference datasets. First of all, researchers need to operate in a adversarial settings [38] where malware authors rapidly adapt their artifacts to new defense mechanisms [12], [25]. This forces the community to detect malware samples through generic techniques instead of careful dissections, with the risk of missing the specific behaviors of new malicious samples [10]. Second, the evaluation of systematic approaches is sensitive to some underlying assumptions which themselves are not grounded. For example, the choice of detection threshold and Anti-Virus engines can largely impact the characteristics of reference datasets [24]. Several flaws can also artificially improve the performance of detectors [2] or mislead the authors about the quality of their output [27].

The goal of our approach is to assist practitioners in the creation of reference datasets. From the result of AV engines, our tool can reconcile the output of these engines and create a vetted reference for other automated techniques.

Prior works on the extraction of AV labels: In the absence of ground truth datasets, practitioners rely on Anti-Virus engines to gather malware labels and cluster them into families [4], [11], [14], [45]. While the collection of malware labels is greatly facilitated by online services such as *VirusTotal* [43], grouping them into families is currently not a straightforward process. The main reason for this difficulty is the lack of widely adopted naming conventions by the industry [9], [10], [26], [29]. In the absence of common standards, end users must deal with the profusion of naming schemes and the lack of consensus among AV labels in malware datasets [7], [24], [31].

To address the issue caused by inconsistent labels, a new tool-supported solution named *AVClass* has recently been proposed by Sebastián et al. [37]. The authors present system able to process scan reports from AV vendors and produce a single label per sample. The authors reported significant performance achievements on a variety of reference datasets [4], [47] in comparison to prior works [7], [31]. However, the authors mention that the system requires a ground-truth list of known malware families to distinguish family names from generic tokens. This limitation goes back to the chicken-and-egg problem that we highlighted earlier. Furthermore, their approach also depends on vendor-specific rules to handle the removal of some vendor suffixes.

In this paper, we address the same challenge as *AVClass*. Nevertheless, we propose an approach tuned to perform well even when labelled samples are not available. We demonstrate that, despite the absence of knowledge on malware families and vendor-specific rules, our approach can compete against *AVClass*.

III. DEFINITIONS

We introduce formal definitions of the terms and concepts that we will refer to throughout the paper.

A. Anti-Virus Labels

Definition 1 (AV Label):
An AV label l is a sequence of words (i.e., alpha-numerical tokens) w_i divided by separators (i.e., blanks and punctuation signs) u_j. Formally, $l = (w_1, u_1, \ldots, u_n, w_{n+1})$.
`Android.Trojan.Adrd.A (B)` is a concrete example of such a label, where '.' (*dot*), '(', ')' (*parentheses*), and '␣' (*space*) are the separators and $Android$, $Trojan$, $Adrd$, A, and B are the words.

Definition 2 (AV Label Field):
A label field f represents the "category" of a given word w_i. The word $Android$ in `Android.Trojan.Adrd.A` (B) indicates the target platform of the malware, while $Trojan$ and $Adrd$ indicates respectively its type and family. Overall, we define 4 fields that match details required in the CARO naming convention [13]: **type** (the kind of threat, i.e., trojan, worm, etc.), **platform** (the OS that the threat is designed to work on, i.e., Windows, Android, etc.), **family** (the group of threats it is associated with in terms of behavior), **information** (extra description of this threat, including its variant).

Grammars described in Tables I and II below provide the lexing rules used by EUPHONY to tokenize AV labels.

Table I: Grammar 1: Lexing rules for EUPHONY

<family>	::=	[:alpha:]{3,}		*<type>*	::=	[:alpha:]{2,}
<info>	::=	[:alnum:]+		*<plat>*	::=	[:alnum:]{2,}
<sep>	::=	([:punct:] | [:blank:])+				

Table II: Grammar 2: Parsing rules for EUPHONY

<word>	::=	<family> | <type> | <plat> | <info>
<label>	::=	<word> | <sep> | <label>

Definition 3 (AV Labeling Pattern):
Given an AV av, its corresponding AV labeling Pattern, noted p_{av} represents the syntax of its labels, i.e., how the different fields are combined to form its labels.

We provide in Table III some illustrative examples of AV labels, their fields and their associated labeling patterns.

Table III: Examples of AV labeling Patterns

Label	AV Pattern	Family	Type	Plat.	Info
Android.Trojan.Adrd	<plat>.<type>.<name>	adrd	trojan	android	–
Trojan:/Adrd.b	<type>:/<name>.<info>	adrd	trojan	–	b
Android.PjApps [Trj]	<plat>:<name> [<type>]	pjapps	trj	android	–
Troj.PjApps (kcloud)	<type>.<name> (<info>)	pjapps	troj	–	kcloud
Android/Adrd.5e2f	<plat>/<name>.<info>	adrd	–	android	5e2f

B. Sample Set

We define a labeling function $label\_of$ which associates a label to a pair (av, app) of AV and application from a dataset:

Definition 4 (Labeling Function):
Let APP be a set of applications, AV a set of AVs, and $\mathcal{L}$ the set of associated labels.

The function $label\_of : AV \times APP \to \mathcal{L}$ maps a pair of AV and application to a label.

From a given label, we further define a family function $family\_of$ which extracts the family field value. More formally:

Definition 5 (Family Function):
Let AV be a set of AVs, APP a set of applications. Let be $\mathcal{L}$ the set of labels such as $\mathcal{L} = label\_of(AV, APP)$, and $\mathcal{F}$ a set of associated family names.

The function $family\_of : AV \times \mathcal{L} \to \mathcal{F}$ maps a pair of AV and label to a family name.

For a given AV, we put together apps with the same family name in a set that we call *Sample Set*. More formally:

Definition 6 (Sample Set):
Let $APP = (app_1, app_2, ..., app_n)$ be a set of app samples, and $\mathcal{F} = (f_1, f_2, ..., f_k)$ a set of associated families. For a given AV av, the sample set S_{av, f_j} defines the set of apps with the same family name f_j. More formally,

$$\forall j \in (1, ..., k),$$
$$S_{av, f_j} = \{app_x \in APP | family\_of(av, app_x) = f_j\}$$

C. Metrics

Sample sets can be disjoint or overlapping, and may be imbalanced. For instance the sample sets represented in the left of Figure 1 are imbalanced as the number of samples associated to the family name f_i by av_a is much smaller than the number of samples associated to family name f_j by av_b. Understanding when a sample set is imbalanced is important when weighting the relevance of a label over another in a dataset.
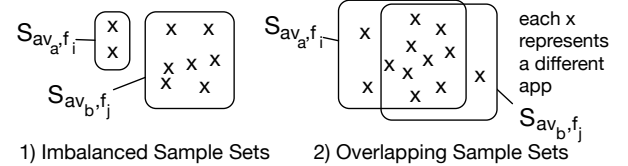

Figure 1: Examples of Sample Sets

Definition 7 (Imbalance Metric):
Given two sample sets S_{av_a, f_i} and S_{av_b, f_j}, we define the *Imbalance* metric as the complement between the minimum and maximum cardinality of the two sample sets, formalized in Equation (1)

$$Im(S_{av_a, f_i}, S_{av_b, f_j}) = 1 - \frac{min(|S_{av_a, f_i}|, |S_{av_b, f_j}|)}{max(|S_{av_a, f_i}|, |S_{av_b, f_j}|)} \quad (1)$$

When both sets S_{av_a, f_i} and S_{av_b, f_j} have the same cardinality, there is no imbalance, and Im is equal to 0. In contrast, imbalance Im gets close to 1 as S_{av_a, f_i} contains fewer apps and S_{av_b, f_j} contains a lot more.

The right part of Figure 1 depicts overlapping sample sets, i.e., a scenario where several apps have been associated at the same time with f_i and f_j by av_a and av_b respectively. This suggests that, despite syntactic differences between both family names f_i and f_j, the family names may characterize the same information (e.g., they point to the same malware). The notion of overlapping is important to assess whether family names should be "merged" together. Thus, we define the exclusion metric which quantifies the degree of overlapping, or lack thereof.

Definition 8 (Exclusion Metric):
Given two sample sets S_{av_a, f_i} and S_{av_b, f_j}, we define the *Exclusion Metric* as the complement of the ratio between 1) the intersection cardinality of two sample sets S_{av_a, f_i} and S_{av_b, f_j} 2) the cardinality of the smallest sample set. This metric is formalized by Equation (2).

$$Ex(S_{av_a, f_i}, S_{av_b, f_j}) = 1 - \frac{|(S_{av_a, f_i} \cap S_{av_b, f_j})|}{min(|S_{av_a, f_i}|, |S_{av_b, f_j}|)} \quad (2)$$

When the sets S_{av_a, f_i} and S_{av_b, f_j} share no app, there is no overlapping as their intersection is empty, and *Ex* is at its maximum at 1. In contrast, as the overlapping gets higher, *Ex* gets close to 0.

Given that family names may contain small syntactic differences, we consider a distance function to relax the equality

constraint on strings. To that end, we define our String distance based on the Sørensen–Dice index [17].

Definition 9 (Distance Metric of Family Names):
The distance metric of family names is computed as the string distance between two family names f_i and f_j:

$$D(f_i, f_j) = 1 - dice(f_i, f_j) \qquad (3)$$

Finally, we measure how two family names f_i and f_j, given by the AVs av_a and av_b respectively, are far to designate the same malware family. More specifically, we compute the distance between two samples sets S_{av_a, f_i} and S_{av_b, f_j} by combining the imbalance, exclusion as well as the string distance metrics that we introduced. The following equation provides the formula that we use:

Definition 10 (Sample Set Distance Metric):
Given two sample sets S_{av_a, f_i} and S_{av_b, f_j}, we define the *Sample Set Distance Metric* as follows:

$$W(S_{av_a,f_i}, S_{av_b,f_j}) = \alpha \times Ex(S_{av_a,f_i}, S_{av_b,f_j}) +$$
$$\beta \times Im(S_{av_a,f_i}, S_{av_b,f_j}) + \gamma \times D(f_i, f_j) \quad (4)$$

where α, β and γ are weight coefficients for adjusting the importance of the different metrics leveraged to compute the sample set distance. First and foremost, we consider that two family names are close to each other only if there is a strong overlap (i.e., low exclusion) between their associated sample sets. Thus, for example, it is not opportune to consider two family names as similar if they do not occur concurrently for the same samples. Consequently, the value of α will reflect the importance of the *Ex* metric. Second, the imbalance of sample sets is considered to account for the degree of granularity within malware families. For example, an AV might assign two family names to a sample set (e.g., $ADRD$, $Pjapps$) while another AV might use only one (e.g., $Pjapps$) family name for all samples in the set. Finally, the impact of typos, which may increase distances between sample sets, requires the string distance to be the least weighted. We have empirically found that a difference of an order of magnitude captures the best relative importance among the coefficients. Thus, in EUPHONY, we set α, β and γ to 1, $\frac{1}{10}$ and $\frac{1}{100}$ respectively.

IV. EUPHONY

Given the lack of consensus among AVs on labeling malicious samples, it is difficult for researchers and practitioners to collect reference datasets of malware [24]. With EU-PHONY [20], we propose to unify the family names assigned by different vendors. We present a summary of EUPHONY's workflow in Section IV-A, and provide details of each step in the following subsections.

A. Overview

Figure 2 illustrates the high-level overview of the architecture of EUPHONY. As input, the tool takes a collection of AV scanning reports. Such reports can be readily obtained from online services such as *VirusTotal* [43], which gathers shared intelligence from a number of AV engines. Then for each sample, EUPHONY performs the following tasks:
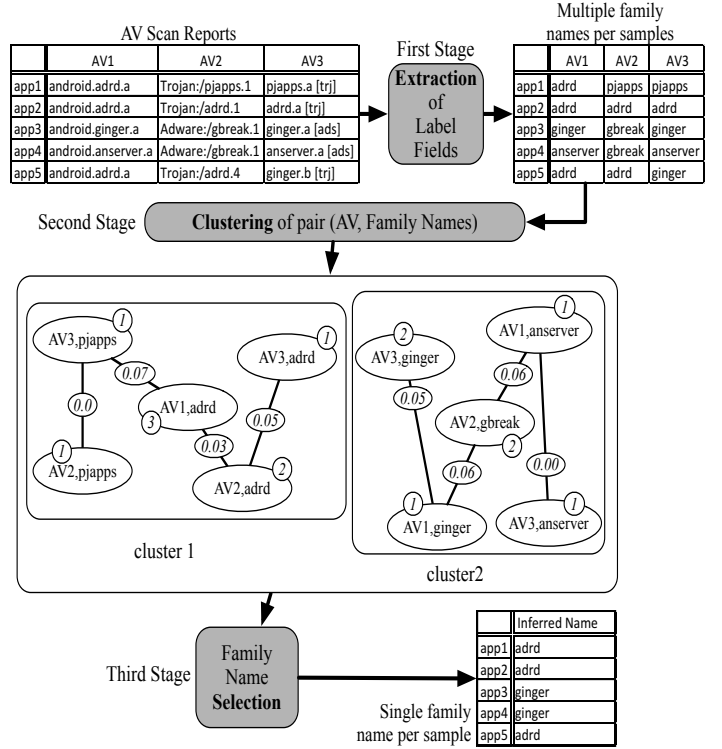

Figure 2: A high-level view of the architecture

- **First stage:** AV labels are pre-processed to derive the family name assigned by each vendor to a given sample. This task allows EUPHONY to deal with different unstructured naming patterns used by the AVs and it is motivated by the lack of convention.
- **Second stage:** This task aims at structuring the relationship between different family names and provides the most appropriate associations between them. EUPHONY analyses both the correlation and the overlap between all family names to understand (i) *mis-labeled* and (ii) *mis-classified* samples. While mis-labeling a sample generally happens when AVs use a different naming scheme for the same family (e.g.: $DroidKungFu$ vs. $DrdKngFu$), mis-classifying a sample is usually associated with a conceptual error made by an AV—with respect to the others (e.g., a vendor labels a sample as $GingerMaster$ while the others decide that belongs to $DroidKungFu$).
- **Third stage** This task aims at bringing consensus between the different vendors and outputs the most appropriate family name for a given sample. Although our framework can also output a set of family names for a sample (i.e.: synonyms), for the sake of simplicity, we only report the most prevalent one.

We next describe the details of each of the component in EUPHONY as well as the choices made during their design and implementation.

B. Extraction of Label Fields from Reports

An AV label is an informally structured string concatenating various pieces of information for describing the malware. In previous section, we have identified 4 recurrent fields in AV
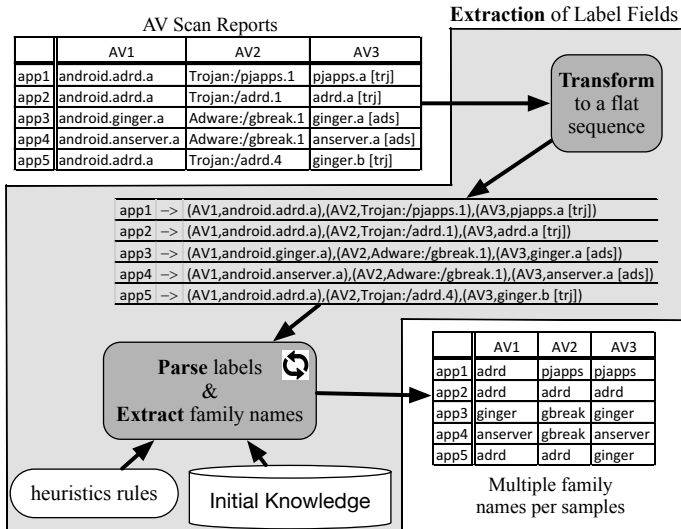
Figure 3: First stage - extraction of label fields from malware reports

AV Scan Reports / **Extraction** of Label Fields

	AV1	AV2	AV3
app1	android.adrd.a	Trojan:/pjapps.1	pjapps.a [trj]
app2	android.adrd.a	Trojan:/adrd.1	adrd.a [trj]
app3	android.ginger.a	Adware:/gbreak.1	ginger.a [ads]
app4	android.anserver.a	Adware:/gbreak.1	anserver.a [ads]
app5	android.adrd.a	Trojan:/adrd.4	ginger.b [trj]

Transform to a flat sequence

app1	->	(AV1,android.adrd.a),(AV2,Trojan:/pjapps.1),(AV3,pjapps.a [trj])
app2	->	(AV1,android.adrd.a),(AV2,Trojan:/adrd.1),(AV3,adrd.a [trj])
app3	->	(AV1,android.ginger.a),(AV2,Adware:/gbreak.1),(AV3,ginger.a [ads])
app4	->	(AV1,android.anserver.a),(AV2,Adware:/gbreak.1),(AV3,anserver.a [ads])
app5	->	(AV1,android.adrd.a),(AV2,Trojan:/adrd.4),(AV3,ginger.b [trj])

Parse labels & **Extract** family names

heuristics rules / Initial Knowledge

	AV1	AV2	AV3
app1	adrd	pjapps	pjapps
app2	adrd	adrd	adrd
app3	ginger	gbreak	ginger
app4	anserver	gbreak	anserver
app5	adrd	adrd	ginger

Multiple family names per samples

label which are identifiable in labels: family, type, platform and extra information. Each vendor generally adopts a specific naming convention to represent and combine these fields in a string. For example, while some vendors start with the platform first, followed by the type and the name; other vendors opt to put the type first or enclose it between square brackets at the end of the string. We further note that AVs change their convention over time, varying field ordering and the punctuation signs that separate fields. In this context, the normalization of their syntax cannot be achieved consistently via fixed rules such as regular expressions.

Another important constraint in parsing AV labels is the lack of a complete, universal and up-to-date lexicon. Indeed, new types, platforms and family names are constantly added by AV vendors to describe emergent threats, and refine the description of old threats. Malware family names in particular are highly dynamic as new malicious behaviors appear regularly.

With these limitations in mind, we propose a number of heuristics for mapping AV label tokens to a lexical field. The overall family name extraction process is described in Figure 3. Given a collection of AV labels, the system can infer the most obvious fields and then iteratively move to the most challenging cases as its knowledge grows. To bootstrap the process, EUPHONY builds on heuristics-based rules, as well as a bare amount of vocabulary on some platform names (e.g., *Android*) and types (e.g., *trojan*). The final output which is the family names given by each AV to the samples are obtained by inferring it from the sample's label for each individual vendor.

1) Parsing algorithm and parameters: The parsing algorithm is at the core of the labeling process and its steps are described in Algorithm 1. The process takes as input a set of labels, some defined heuristics and an initial knowledge database on malware lexicon. First, the algorithm tokenizes each AV label and initializes the mappings between the tokens and the different label fields. At this stage, a given token can be associated to all fields (name, type, platform or information). To decide the unique field to which it should

---

Algorithm 1 Incremental Parsing by EUPHONY

```
 1:  Mapping ← [name, type, platform, information]
 2:  function PARSE(knowledge_db, heuristics, labels)
 3:      mappings ← MAP(Mapping, labels)
 4:      pqueue ← PRIO-QUEUE(mappings)
 5:      while NOT-EMPTY?(pqueue) do
 6:          m ← PEEK(pqueue)
 7:          for H in heuristics do
 8:              findings ← H(knowledge_db, m)
 9:              MERGE(m, findings)
10:          end for
11:          if COMPLETE?(m) then
12:              ENRICH(knowledge_db, m)
13:          else
14:              PUSH(pqueue, m)
15:          end if
16:      end while
17:      return mappings
18:  end function
```

be assigned, the algorithm proceeds by iteratively eliminating improper assignment, starting with the easiest cases: the order of processing is conveyed by a priority queue (line 4), where mappings with the least amount of unknown fields are pushed at the head of the queue, while mappings with the most amount of unknown fields are pushed back at the end of the queue. At each step, the algorithm takes the first mapping of the queue (line 6) and applies the heuristics-based rules to collect more information about the mapping (line 8). Then, it merges this information to create a new mapping. In case of conflicts, the merge operation will always keep the oldest knowledge at its disposal.

If the mapping is complete at the end of this operation (line 11), i.e. if each token is associated to a single field, this mapping is removed from the queue and its information pieces are extracted to enrich the knowledge database (line 12). Otherwise, the mapping is pushed back in the queue to be processed at a later iteration (line 14). Once the queue is empty, the mapping list is returned with the complete list of associations (line 17). To force early termination, EUPHONY provides a parameter for setting a maximum number of iterations performed by the algorithm.

2) Heuristics rules and initial database: We now provide details on the parameters of the algorithm. In our current implementation, we rely on 10 heuristic rules to find associations between words and fields. These rules are listed in Table IV.

	Property	Action
1	Word is associated to a known field in the database	associate the same field
2	Word is suffixed by -ware	word is a type
3	Word is between parenthesis	word is an info
4	Word is between square brackets	word is a type or info
5	Only one family, type and platform per label	enforce when field is found
6	Word is a synonym of a type or platform in the database (e.g. troj, trojan)	associate the same field
7	Word is the last token not associated to a field	word is a family
8	Words are part of common word sentence	words are info
9	Label is compatible with a pattern of the same AV	associate fields based on pattern
10	Given two remaining tokens, one is a common word and the other is not	common word: information, other word: family

Table IV: Heuristics for mapping words to fields.

Let's consider two of the rules to illustrate the associated action. Rule 1 is the most straightforward heuristic. During its execution, EUPHONY accesses the database to check if the word is already associated to a particular field. In particular, the word *Android* is commonly known to match with the field `platform`. Thus, Rule 1 can leverage existing knowledge to

identify obvious fields. Rule 9, on the other hand, is tuned to create more knowledge by inferring the field of an unknown token. For example, given the AV label 'ransom.android.pjapps' and its incomplete mapping [ransom: ?, android: `platform`, pjapps: `family`], the algorithm can know at this stage that *ransom* is likely a `type`, and yield the following AV labeling pattern: '`<type>.<platform>.<name>`'. This inference is validated by correlating with mappings for all samples of the same AV. Once the mapping is complete, the inferred information will be added to the database and support the identification of more tokens.

To bootstrap the inference process, our algorithm requires an initial lexicon about malware labels. Generally, a small but widely accepted lexicon can be found online in specialised knowledge bases. We stress that, in EUPHONY, such a lexicon does not have to be exhaustive for our algorithm to work properly. For example, in our experimental setting, we have leveraged a limited lexicon including only most well know types, platforms and information enumerated by the Microsoft Malware Protection Center[2]. Table V provides statistics and examples of tokens contained in this list. In particular, we observed that important words such as "Android", "Malware" or family names are not present. We demonstrate the automated nature of the inference system by relying only on this available lexicon without any modifications of its entries.

Field	# of Entries	Example
TYPE	34	adware, backdoor, spyware, trojan, worm
PLATFORM	74	linux, androidos, iphoneos, java, win32
INFORMATION	18	dll, rootkit, plugin, pak, gen

Table V: Initial database entries

C. Grouping of Similar Labels

After parsing malware labels to identify family names given by different AVs to each sample, EUPHONY builds a graph representing the association links between family names based on their assignment on same samples. Then, based on a threshold parameter that determines the granularity of grouping, clusters of family names are separated. Figure 4 provides an overview of the process. In the rest of the paper, we will often use the terms "name" instead of the full expression "family name".

1) Associating family names: At the end of the previous stage, EUPHONY has a new dataset where each sample is associated with multiple malware family names reported by AVs. These potentially syntactically different names may include mis-labeling noises and mis-classification errors, which make the process of selecting unique names more difficult.

We study the associations between AV family names to group together commonly related names. We found that the most natural method to analyze potential associations was to construct a weighted graph $G = (N, E)$, where a node $n \in N$ represents a name that an AV assigned: $n = (av_a, name)$, and an edge $e = [(av_a, name_x), (av_b, name_y)] \in E$ indicates that both AVs av_a and av_b have labeled a same sample with $name_x$ and $name_y$ respectively. From the information attached to a
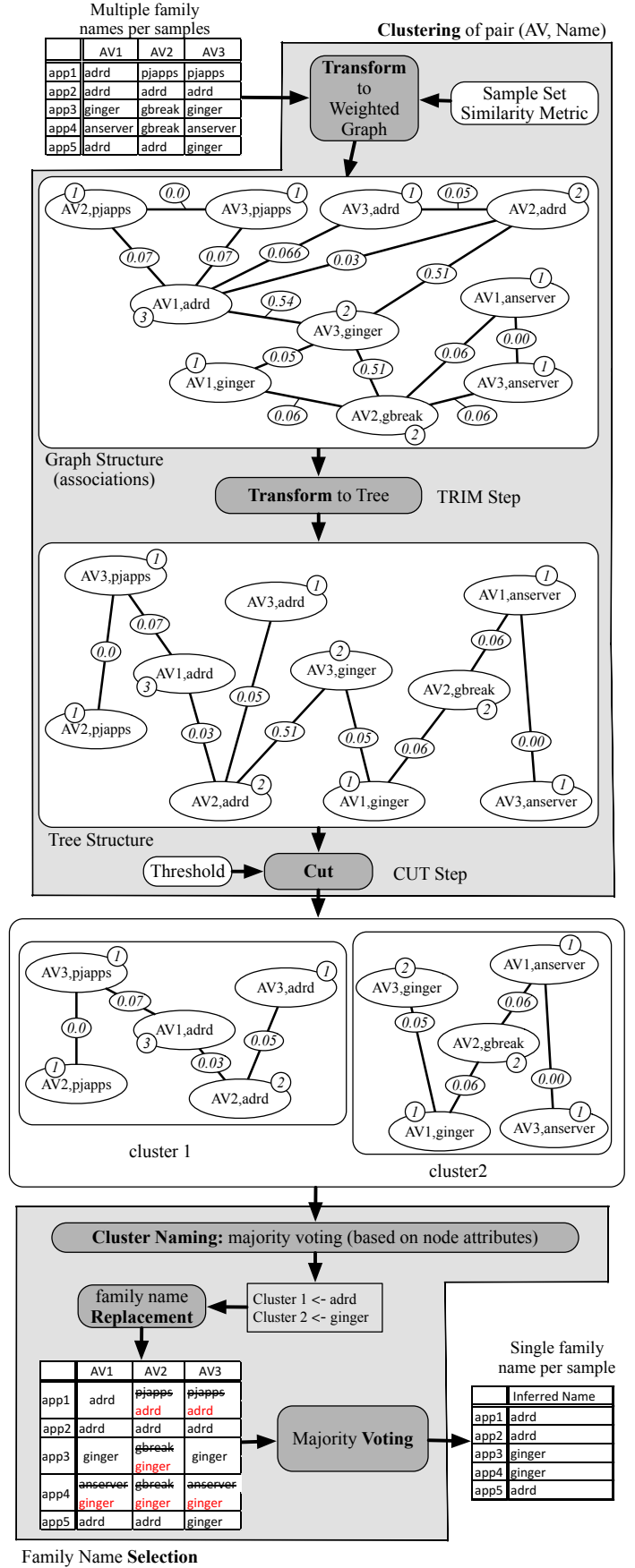
[2]Malware Naming Conventions: http://bit.ly/2f3vKlu

Figure 4: Second stage - Clustering & Third stage - Family Name Inference

node, EUPHONY can identify the corresponding Sample Set $S_{ava,name}$ and computes, for each node, the size of its sample set (i.e., the number of times the family name is given by that AV in the dataset). It also computes, for each edge, the overlapping between the related sample sets (i.e., the number of times AVs agree on the name given at that node). Both pieces of information are then used to compute the *Imbalance* and *Exclusion* metric, and eventually the *Sample Set Distance* metric, also taking into account the string distance between family names, as defined earlier in Section III-C. Imbalance and Exclusion metric values are used as edge attributes in the graph, while the Sample Set Distance metric value is used as the edge weight (in Figure 4 only the edge weight is represented). Note that the weight of an edge represents the degree to which the sample sets associated to the connected nodes are dissimilar: the lower the weight, the more likely both sample sets belong to the same cluster.

2) Grouping family names: Given the large number of associations that we have observed among family names in our datasets, we expect the weighted graph to be highly connected and thus include very few identifiable sub-graphs. For instance, a generic name can create additional edges with more specific names and thus tie together components that were otherwise weakly related. Processing mistakes during label fields extraction can also introduce fake associations among family names. It is therefore important to remove such undesired associations from the graph and only keep sub-graphs with strongly related nodes. To that end, we build a technique that comprises two successive steps, referred to as TRIM and CUT:

- In the TRIM step, we use Prim's algorithm [32] to transform the graph into a Minimum Spanning Tree (i.e., the sum of its edge weights is minimum). The goal of this operation is to reduce the complexity of the original structure and keep the most similar edges as long as they do not introduce cycles in the graph. The complexity of this algorithm is $O(|E|\log|N|)$ in our current implementation.
- The CUT step takes the tree structure and applies a filter function to remove edges whose weight exceed a given threshold value. As a result, the input tree is divided into connected components that can be interpreted as clusters of strongly related family names. The complexity of this algorithm is $O(|E|)$ in our implementation.

D. Inference of Family Name per Sample

In the last stage, we study the relation between the different family names to assess the prevalence of predominant naming schemes used by the different AVs. In particular, we created a list of associations that put in relation all family names within the cluster where they are grouped together. More specifically, we first associate a single name to each cluster. This name is inferred as the most frequent name present in the cluster by taking into account the attribute of each node. In the step illustrated at the bottom of Figure 4, cluster 1 is associated to $adrd$ and cluster 2 to $ginger$. Then, each family name present

Table VI: Datasets used in our evaluation

Reference	Wild	Samples	Families	Anti-Virus	Collection Period
MalGenome [47]	✗	1 262	44	58	08/2008 - 10/2010
Drebin [4]	✗	5 260	178	57	08/2010 - 10/2012
Androzoo [3]	✓	402 600	unknown	63	01/2015 - 08/2016

in a cluster is replaced by the cluster name. For instance, in Figure 4, $pjapps$ given by $AV2$ is replaced by $adrd$. More generally, in this example, all occurrences of $pjapps$ are replaced by $adrd$ and all occurrences of $anserver$ or $gbreak$ by $ginger$.

Once the replacements are done, to infer a single family name per sample, we implement a majority voting where we compute the frequency of each family name and select the most frequent one per sample. In case of a tie, we use the highest frequency of the names within the dataset to choose between the candidates and break the conflict.

V. EVALUATION

This section reports EUPHONY's results on Android malware. We first evaluate the performance of EUPHONY against two reference datasets widely used in the literature. Next, we investigate the potential of our system when confronted with large scale scenarios of malware found in the wild. For the sake of completeness, we describe the dataset and metrics used and we compare our results against the state-of-the-art.

A. Datasets and Metrics

The evaluation of EUPHONY is based on two different sets of samples: (i) *reference datasets*, and (ii) an *in the wild dataset*. We next describe the source of each of the datasets used (see Table VI for a summary) and the metrics used to evaluate our approach.

Reference datasets: These datasets have been distributed by the research community together with a reference ground truth of malware families, and have been widely used in the literature recently [4], [16], [28]. For our study, we consider *MalGenome* [47], a dataset manually vetted and collected between 2008 and 2010, and which includes 1 262 samples regrouped into 44 families. Similarly to previous work [37], we update this dataset by grouping into a single family all variants of $DroidKungFu$ ($DroidKungFu1$, $DroidKungFu2$, $DroidKungFusApp$, etc). Additionally, we also consider *Drebin* [4], a dataset collected between 2010 and 2012, and which includes all samples from *MalGenome* as well as an additional set of 3 998 more samples. *Drebin* includes 178 families.

In-the-wild dataset: We collected recent samples from *Androzoo* [3], a repository that shares samples from a variety of sources as well as their AV labels provided by *VirusTotal*. For our study, we leveraged the public download API and retrieved 402 600 samples created between January 2015 and August 2016[3]. We ensured that all samples were classified as malware by at least one AV[4].

---

[3]*Androzoo* bases its timeline on the DEX compilation date.

[4]Overall, the samples were labeled by 63 AVs

Evaluation metrics: Let S be a sample dataset, $G = \{G_1, \ldots, G_s\}$ be the set of s "ground-truth" clusters from S, and $C = \{C_1, \ldots, C_n\}$ be the set of n clusters output by a given tool over S. Similarly to previous work [37], we define the following metrics:

- **Precision:** $Prec = \frac{1}{n} \times \sum_{j=1}^{n} max_{k=1,\ldots,s}(|C_j \cap G_k|)$
- **Recall:** $Rec = \frac{1}{s} \cdot \sum_{k=1}^{s} max_{j=1,\ldots,n}(|C_j \cap G_k|)$
- **F1-score:** $F1 = 2 \times \frac{Prec \times Rec}{Prec + Rec}$

While precision measures the effectiveness of a tool to map outputted clusters into ground-truth clusters, recall quantifies the effectiveness of the tool to map ground-truth clusters into outputted clusters. Finally, the F-Measure represents the harmonic mean between precision and recall.

For the purpose of this paper, we first investigate the precision and recall reported when clustering malware samples in the reference datasets. This allows us to quantitatively compare our approach with previous work [37]. We then use the samples collected in the wild to evaluate a number of statistical metrics such as the number of families, the number of singletons and the most relevant labels.
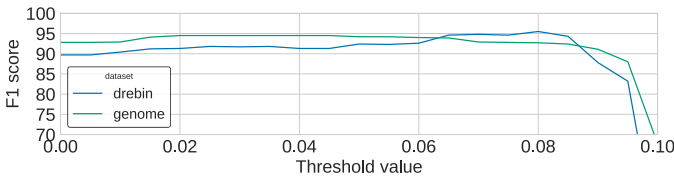
B. Performance Evaluation

Figure 5: Parameter selection for threshold value

EUPHONY uses a threshold to control the clustering sensitivity by breaking edges whose weight exceeds the given value. On Figure 5, we observed that a threshold of 0.07 represented a good trade-off between nose-reduction and accuracy.

We now evaluate the performance reported over the reference datasets (see Section V-A). The leftmost part of Table VII shows the precision, recall and F1 measure for EUPHONY. Results show that clustering *MalGenome* is, overall, more challenging than clustering *Drebin*, with a F1 measure of 92.7% and 95.5% respectively. This can be explained by looking at the precision, which shows that not all predicted clusters can be mapped to their respective reference clusters. Interestingly, the score obtained for the recall indicates that almost all referenced clusters (i.e., 99.7% of the malware families) in *MalGenome* have been correctly predicted. Instead, only 96.1% of the referenced clusters in *Drebin* have been correctly predicted.

When analyzing *MalGenome* results, we observe that some AVs prefer to combine together some reference clusters to form one single super-family. For instance, families $ADRD$ (22 samples) and $Pjapps$ (58 samples), are perceived as one large family called $Pjapps^*$ (with 80 samples). Similarly, $BaseBridge$ (122 samples) and $AnserverBot$ (187 samples) are perceived as $Basebridge^*$ (with 309 samples). This is actually understandable as authors in [47] believe that $AnserverBot$ actually evolved from $BaseBridge$, inheriting common features. Other recent works have also confirmed this

and pointed out that some other families are also strongly related to each other [41]. Based on this, one can conclude that the perception of the AVs is, in some general cases, more coarse grained. Thus, they can treat two similar clusters as one single family. We refer the reader to Section VI for further discussions.

As for the results obtained with *Drebin*, we found some cases where the AVs agree on using a more fine-grained definition of some families than the one given in the reference ground-truth. For example, most of the samples from the reference family $Opfake$ (952 samples) are subdivided into two sub-families, i.e.: $Opfake$ (546 samples) and $SMSSend$ (25 samples). Similarly, most of the samples in $ExploitLinuxLotoor$ (70 samples) are sub-divided into three sub-families: $Lotoor^*$ (58 samples), $GingerBreak$ (9 samples) and $AsRoot$ (8 samples).

Comparison against the state-of-the-art: To provide further insights on the performance achieved by EUPHONY, we compare our results with *AVClass* [37]. Since the authors have analyzed their approach on *MalGenome* and *Drebin* datasets, we use the results in their paper as our evaluation baseline (Table VII, *AVClass* Config 1). We also replicate their experiments by taking into account the difference between the inputs requirements of both tools. On one hand, EUPHONY uses a small list of some well-known words (not including family names) about malware labels. On the other hand, *AVClass* requires a list of malware families to construct a set of *generics tokens* and *aliases*. These sets are built in a two-step process. First, *AVClass* uses the list of malware families to distinguish these family tokens from other so-called *generics tokens* (e.g. types, platforms, information, ...). Second, *AVClass* strips generics tokens from malware labels to discover *aliases* among malware family names. Finally, the sets of *generics tokens* and *aliases* are leveraged to produce the final output of the tool: a single family name per malware sample using a plurality voting. Several factors may thus impact the performance of AVClass, including i) the exhaustiveness of the inputted list of malware families, and ii) the error rate in the generation of the sets of generics and aliases. Consequently, we consider three different scenarios for our evaluation:

- An updated version of *AVClass*—we use the last version of the tool released on GitHub [6], taking into account recent code fixes, as well as updates to complete the list of generic terms and aliases [5]. The results are reported in Table VII, *AVClass* Config 2.
- An automatic inference of *aliases* only—we use the authors script with the default settings to generate a list of aliases based on our two reference datasets (Table VII Config 3).
- An automatic inference of both *generics* and *aliases*— we use the authors scripts with the default settings on the union of our two reference datasets to build the knowledge necessary to *AVClass*'s functioning (Table VII Config 4).

All results are reported in Table VII. For all four configurations, EUPHONY performs better than *AVClass* in terms

Table VII: Performance of EUPHONY and comparison against the State-of-the-art (in %)

| | EUPHONY | | | | AVClass | | | | | | | | | | |
| | - - | | | | AVClass Config 1 as reported in [37] | | | AVClass Config 2 with default files in Git | | | AVClass Config 3 new aliases only | | | AVClass Config 4 new generics & aliases | | |
Dataset	Prec	Rec	F1	#	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1
MalGen.	86.7	99.7	92.7	33	87.2	98.8	92.6	86.5	98.0	91.9	86.2	99.0	92.1	53.9	65.2	59.0
Drebin	95.0	96.1	95.5	142	95.2	92.5	93.9	95.4	93.0	94.2	95.6	90.6	93.0	29.6	69.8	41.6

of F1 score (harmonic mean between precision and recall). Nonetheless, we observed that their precision is comparable to ours in those configurations where prior knowledge (aliases and generic terms) is provided. When we inferred both (i) aliases, and (ii) aliases and generic terms using the samples given in the reference datasets (i.e.: *MalGenome* and *Drebin*)[5], we observed that the performance drops drastically. Typical families included in Config. 4 are: android (537 samples), trojan (377 samples) and basebridge (68 samples). This shows that the performance of *AVClass* is driven by the input of an initial knowledge—which should be collected by the final user. In contrast, EUPHONY does not require any guiding process and/or pre-defined knowledge of the families. The only prior knowledge required by our framework is a basic understanding of some common types of malware (i.e., trojan, virus, etc.), execution platforms (i.e., android, linux, Win32, etc) and information (e.g. dll, pak, gen).

C. Evaluation on Samples in the Wild

This section reports our experiments in the wild. We analyze the number of samples that EUPHONY can group together with respect to *AVClass*. Note that in this section we report results using the same experimental setting used above (see Appendix **??**). As for *AVClass*, we choose the most favorable configuration[6]. Table VIII summarizes the results obtained on the *Androzoo* dataset.

Table VIII: Results for *Androzoo* (402 600 samples)

	Labeled	Clusters	Singletons	Runtime
EUPHONY	319 100	735	165	216s
AVClass	178 471	453	135	114s

Results show that EUPHONY managed to cluster 79% of the samples (319 100 out of 402 600). In contrast, *AVClass* clustered 44% (178 471 out of 402 600). This means that a practitioner using *AVClass* would not obtain labels for more than half of the recent dataset. This can be partly explained by the strategy used by *AVClass* to handle generic terms in labels, as well as aliases in family names. On the contrary our approach does not present distinctions between generic and specific malware families, and may thus find more associations. This can further provide a better understanding of the appropriate set of samples in every cluster. In this regard, EUPHONY has split the dataset into 735 clusters and produces 165 clusters with one single sample (namely, singletons).

[5]Note that *AVClass* published results [37] were obtained using knowledge on aliases and generics that was built from larger datasets

[6]This is, using the default lists of aliases and generics collected by the authors. These lists may have been manually improved to guarantee the labeling system out of the lab

Table IX: TOP 10 clusters EUPHONY & AVClass

EUPHONY		AVClass	
Family	Samples	Family	Samples
dowgin	37 739	kuguo	38 532
kuguo	25 005	dowgin	22 643
addisplay	20 862	secapk	20 492
jiagu	20 705	airpush	13 209
anydown	19 621	jiagu	8 987
secapk	18 224	smsreg	8 427
generic	17 836	feiwo	7 399
agent	17 596	revmob	6 376
inmobi	16 203	leadbolt	5 348
airpush	13 267	anydown	5 147

Instead, *AVClass* proposed 453 clusters and 135 singletons. Note that the runtime overhead of EUPHONY is negligible compared to *AVClass*, even in this setting where the creation of *generics tokens* and *aliases* for *AVClass* was skipped.

Table IX shows the Top 10 clusters (in terms of size) for both EUPHONY and *AVClass*. Both approaches report clusters of the same order of magnitude, and with similar family names. This indicates that EUPHONY reach to similar conclusions than *AVClass* for the most popular families, but without prior knowledge on generic families. We can also observe that our approach can deal with generic AV labels. For instance, a common field used by AVs is "trojan.androidos.generic.a", with 94 255 occurrences (4%). We can further observe 576 261 occurrences (23%) of the string "gen" in the list of labels. This contrast with the most occurring family, $Dowgin$, with 311 593 times (14%). This explains the lack of coverage reported by *AVClass*. We position here that being aware of theses types of clusters are important to filter out samples that might interfere with the proper classification of other clusters. In practice, adware and other type of grayware [40] could be identified. Nevertheless, to account for corner cases where a generic term is selected as a family name, practitioners can inject into EUPHONY their knowledge on how a specific tokens must be associated to label field. Since a significant portion of samples remained unlabeled by *AVClass* and EUPHONY, we only consider the subset of samples that are labeled by both tools to investigate the similarities and differences among reported clusters. We provide in Table X the statistics on the new Top-10 clusters (dropping samples that are unlabeled by either tool) and information on the extent to which they overlap. Most top families strongly overlap. For example, $Dogwin$, the most prevalent family in EUPHONY, overlaps with the also-labeled $Dogwin$ family in *AVClass* with a ratio of 95%. For 7 of the top-10 clusters given by EUPHONY, we find that the corresponding cluster by *AVClass* overlaps at over 85%. If we take the particular case of samples labeled as $Kuguo$ by *AVClass*, EUPHONY splits them into mainly three families ($kuguo$, $addisplay$, $hiddeninstall$) with an overlap of 99%, 54% and 93% respectively.

Table X: TOP 10 clusters of EUPHONY compared to *AVClass*

EUPHONY		AVClass		Intersection	
Family	samples	Family	samples	samples	overlap (in %)
dowgin	33 297	dowgin	22 617	21 035	93.0
kuguo	24 273	kuguo	38 532	24 072	99.2
secapk	17 889	secapk	20 492	17 825	99.6
addisplay	11 203	kuguo	38 532	6 055	54.0
airpush	10 055	airpush	13 202	10 017	99.6
jiagu	7 215	jiagu	8 987	7 211	99.9
smsreg	6 294	smsreg	8 427	5 819	92.5
agent	6 088	feiwo	7 399	1 014	16.7
revmob	6 061	revmob	6 376	6 058	99.9
generic	5 663	anydown	5 147	1 890	36.7

VI. DISCUSSION

Threat intelligence sharing is essential to further research on malware detection and classification. Online services such as *VirusTotal* already contribute to this effort by providing diversified AV labels for any uploaded sample. Building reference datasets, either manually or leveraging AV labels, remains challenging. With EUPHONY [20], we propose an approach for automatically building such datasets by inferring sample labels from a set of labels provided by non-consensual AVs. EUPHONY improves over *AVClass* by overcoming its main limitations of requiring a substantial initial knowledge on malware families and antivirus vendors to bootstrap the labeling process.

A. Use Case in the wild

As new malware families appear, reference datasets need to be regularly updated with no a-priori knowledge on the aliases and generic terms that AV vendors now use to label samples. From a small and relatively stable list of common tokens, EUPHONY can (1) infer missing information on tokens in AV label strings using heuristics, and (2) group similar families together according to a comprehensive distance metric that takes into account typos in naming, imbalance in label assignment among AV sample sets, and overlapping of sets.

While EUPHONY can be used off-the-shelf, without any requirement of expertise on malware labels, advanced users may also build on top of our framework and specify their own heuristics, metrics or knowledge of malware labels.

B. Familial Ties

As it can be observed on *MalGenome* and *Drebin*, family names extracted from AV labels suggest that some clusters overlap, indicating ties among families. Indeed, a reasonable proportion of samples from a given reference cluster C_x may be labeled by AVs using a family name associated to another reference cluster C_y[7]. Such familial ties are challenging to properly address by a labeling system, as they indicate that there may be a hierarchy in families, with super-families regrouping others. EUPHONY handles the case of family ties by taking into account the imbalance in label occurrences within sample sets to decide on the opportunity to merge them. The threshold value, on the one hand, further controls the sensitivity of the clustering process, but, on the other hand, can lead to more labeling errors which EUPHONY cannot estimate on in-the-wild datasets.

---

[7]Out of the 7 504 AV labels assigned to samples in the $AnserverBot$ folder of *MalGenome*, 2 575 actually use $*BaseBridge*$ as the family name, which however is the name of another folder in *MalGenome*

C. Towards a Better Ground-truth

The creation of better ground-truth datasets is an important objective to support the development and the deployment of new machine-learning based systems. In future work, we would like to provide a clearer interpretation of the relation between malware families and their structural features. In particular, we are planing to analyze the prevalence of features across families. Recent approaches [39] have shown to be very efficient at identifying syntactic and resource-centric features that can characterize Android malware. We position that adding these features as input to our algorithm—and in addition to the AV labels—could contribute to alleviate disagreements among AV vendors. We also intend to compare this against traditional clustering systems that would not consider AV labels to understand to what extent labels are essential artifacts.

VII. CONCLUSION

In this paper, we presented the design and implementation of EUPHONY [20], an automated labeling system for Android malware. This approach is the first attempt to break the vicious circle which requires practitioners to possess a well-studied reference of malware to operate on new and unknown samples. Thus, we hope to lift an important limitation that impers the development of the most recent auto-learning techniques.

EUPHONY can operate with minimum knowledge on malware label lexicon and infer malware families with a high precision. Moreover, it can handle inconsistencies observed in Anti-Virus labels and allow a fine tuning of the granularity of families.

We contribute to the community's effort towards building reliably assessed approaches, by establishing a long-needed framework for regularly updating reference datasets with most recent samples, retrieved via the public API of shared repositories such as Androzoo [3], that we automatically label with EUPHONY.

REFERENCES

[1] Y. Aafer, W. Du, and H. Yin. Droidapiminer: Mining api-level features for robust malware detection in android. *Security and Privacy in Communication Networks*, 127:86–103, 2013.

[2] K. Allix, T. F. Bissyandé, Q. Jérome, J. Klein, R. State, and Y. Le Traon. Empirical assessment of machine learning-based malware detectors for android. *Empirical Software Engineering*, 2014.

[3] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: collecting millions of android apps for the research community. In *Proceedings of the 13th International Workshop on Mining Software Repositories*, pages 468–471. ACM, 2016.

[4] D. Arp, M. Spreitzenbarth, H. Malte, H. Gascon, and K. Rieck. Drebin: Effective and explainable detection of android malware in your pocket. *Symposium on Network and Distributed System Security (NDSS)*, pages 23–26, 2014.

[5] AVClass. Avclass repository cloned on oct 24, 2016. commit head: 80c14adcc29978ab813b41c73dd485072e576140.

[6] AVClass. Github of avclass. https://github.com/malicialab/avclass.

[7] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of internet malware. *Recent Advances in Intrusion Detection*, 4637/2007:178–197, 2007.

[8] D. Barrera, H. G. ü. b. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. *Proceedings of the 17th ACM CCS*, (1):73–84, 2010.

[9] V. Bontchev. Current status of the caro malware naming scheme. *Virus Bulletin (VB2005), Dublin, Ireland*, 2005.

[10] P.-M. Bureau and D. Harley. A dose by any other name. In *Virus Bulletin Conference, VB*, volume 8, pages 224–231, 2008.

[11] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: Behavior-based malware detection system for android. *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices - SPSM '11*, (January):15, 2011.

[12] J. Canto, H. Sistemas, M. Dacier, E. Kirda, and C. Leita. Large scale malware collection: lessons learned. *27th International Symposium on Reliable Distributed Systems.*, 52(1):35–44, 2008.

[13] CARO. Caro naming convention. http://www.caro.org/articles/naming.html.

[14] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. Mast: triage for market-scale mobile malware analysis. In *Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks*, pages 13–24. ACM, 2013.

[15] W. Chen, D. Aspinall, A. D. Gordon, C. Sutton, and I. Muttik. More semantics more robust: Improving android malware classifiers. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 147–158. ACM, 2016.

[16] S. K. Dash, G. Suarez-Tangil, S. Khan, K. Tam, M. Ahmadi, J. Kinder, and L. Cavallaro. Droidscribe: Classifying android malware based on runtime behavior. In *Mobile Security Technologies (MoST 2016)*, 2016.

[17] L. R. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, 1945.

[18] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX Security*, page 21, 2011.

[19] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. *Proceedings of the 16th ACM conference on Computer and communications security - CCS '09*, pages 235–245, 2009.

[20] Euphony. Github of euphony. https://github.com/fmind/euphony.

[21] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[22] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54. ACM, 2013.

[23] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker : Scalable and accurate zero-day android malware detection categories and subject descriptors. *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, pages 281–293, 2011.

[24] M. Hurier, K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. On the lack of consensus in anti-virus decisions: Metrics and insights on building ground truths of android malware. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, DIMVA 2016, pages 142–162, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[25] A. Kantchelian, M. C. Tschantz, S. Afroz, B. Miller, V. Shankar, R. Bachwani, A. D. Joseph, and J. D. Tygar. Better malware ground truth: Techniques for weighting anti-virus vendor labels. AISec '15, pages 45–56. ACM, 2015.

[26] T. Kelchner. The (in)consistent naming of malcode. *Computer Fraud and Security*, 2010(2):5–7, 2010.

[27] P. Li, L. Liu, D. Gao, and M. K. Reiter. *Recent Advances in Intrusion Detection: 13th International Symposium, RAID 2010. Proceedings*, chapter On Challenges in Evaluating Malware Clustering, pages 238–255. Springer Berlin Heidelberg, 2010.

[28] M. Lindorfer, M. Neugschwandtner, and C. Platzer. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *COMPSAC'14*, 7 2015.

[29] A. Marx and F. Dessmann. The wildlist is dead , long live the wildlist ! table of contents. (SEPTEMBER):136–146, 2007.

[30] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the 2012 ACM CCS*, pages 241–252. ACM, 2012.

[31] R. Perdisci and M. U. Vamo: towards a fully automated malware clustering validity analysis. *Annual Computer Security Applications Conference*, page 329, 2012.

[32] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.

[33] V. Rastogi, Y. Chen, and W. Enck. Appsplayground : Automatic security analysis of smartphone applications. *CODASPY '13 (3rd ACM conference on Data and Application Security and Privac)*, pages 209–220, 2013.

[34] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EuroSec, April*, 2013.

[35] C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. Van Steen. Prudent practices for designing malware experiments: Status quo and outlook. *Proceedings of S&P*, pages 65–79, 2012.

[36] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Android permissions: a perspective combining risks and benefits. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, SACMAT '12, pages 13–22, New York, NY, USA, 2012. ACM.

[37] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 230–253. Springer, 2016.

[38] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Proceedings of the 2010 IEEE S&P*, pages 305–316.

[39] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro. DroidSieve: Fast and accurate classification of obfuscated android malware. May 2017.

[40] G. Suarez-Tangil, J. E. Tapiador, P. Peris, and A. Ribagorda. Evolution, detection and analysis of malware for smart devices. *IEEE Communications Surveys & Tutorials*, 16(2):961–987, May 2014.

[41] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco. Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Systems with Applications*, 41(4):1104–1117, 2014.

[42] S. University of Luxembourg. Androzoo repository. https://androzoo.uni.lu.

[43] VirusTotal. VirusTotal about page. https://www.virustotal.com/en/about/.

[44] L. Weichselbaum, M. Neugschwandtner, M. Lindorfer, Y. Fratantonio, V. van der Veen, and C. Platzer. Andrubis: Android malware under the magnifying glass. *Vienna University of Technology, Tech. Rep. TRISECLAB-0414*, 1:5, 2014.

[45] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*, pages 62–69, 2012.

[46] L. Yan and H. Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. *Proceedings of the 21st USENIX Security Symposium*, page 29, 2012.

[47] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE S&P*, pages 95–109, 2012.

[48] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, (2):5–8, 2012.