

# Finding Non-trivial Malware Naming Inconsistencies

Federico Maggi    Andrea Bellini    Guido Salvaneschi    Stefano Zanero

Technical Report TR-2011-  
Dipartimento di Elettronica e Informazione, Politecnico di Milano

**Abstract** Malware analysts, and in particular antivirus vendors, never agreed on a single naming convention for malware specimens. This leads to confusion and difficulty in comparing coverage of different antivirus engines, and generally causes issues, in particular for researchers, when integrating and systematizing known threats, or comparing the outcome of different detectors. Given the observation that solving naming inconsistencies is almost an utopia—because it would require all the vendors to agree on a single naming convention—in this paper we take a step back and concentrate on the problem of finding inconsistencies. Solving inconsistencies is indeed impossible without knowing exactly where they are. To this end, we first represent each vendor’s naming convention with a graph-based model. Second, we give a precise definition of inconsistency with respect to these models. Third, we define two quantitative measures to calculate the overall degree of inconsistency between vendors. In addition, we propose a fast algorithm that finds non-trivial (i.e., structural) inconsistencies. Our experiments on four major antivirus vendors and 98,798 real-world malware samples confirm anecdotal observations that different vendors name viruses differently. More importantly, we were able to find inconsistencies whose existence cannot be inferred at all by looking at the naming convention characteristics (e.g., syntax).

## 1 Introduction

The current threat landscape is characterized by self-replicating malicious code written by notoriety-driven authors gave way to money-driven malware campaigns [5] spread through drive-by download [6]. In addition, classic polymorphic viral engines gave way to multiple layers of packing, obfuscation, recompilation, and advanced self-update mechanisms. As a consequence, a rising number of unique malware specimens, often (slightly) mutated versions of known malware, spurred a partial transformation in the mechanisms of action of antiviruses, which rely more and more on generic signatures and heuristics [9]. As a result, naming and detection have become twisted.

For historical reasons, malware naming has never followed established convention [9]. In fact, antivirus companies and researchers used to name viruses based on characteristic they found interesting. However, naming inconsistencies become a real research problem when trying to correlate or mine useful data across different antiviruses. Even simple problems such as comparing “top ten” threat lists turn to be very difficult<sup>1</sup>. Researchers have concentrated on the problem of solving inconsistencies and proposed both pragmatic approaches (e.g., VGreep, Wild List [1]), or naming conventions (e.g. CARO [3]).

In this paper we underline the importance of *finding* naming inconsistencies before concentrating on methods for *solving* them. To this end, we extend the notion of “consistency” between naming conventions presented in literature, and propose a systematic

<sup>1</sup> <http://infosecurity-us.com/view/6314/malware-threat-reports-fail-to-add-up>

approach to identify and quantify the discrepancies between classification of specimens by different antivirus engines. Our goal is finding where the inconsistencies lie. Starting from these “hot spots” and armed with the knowledge of a vendor’s detection methodology, an expert can leverage our mechanism to investigate about the causes of such inconsistencies. Our approach provide guidance at deriving such causes, as opposed to finding them automatically, which would require to model the detection process of each vendor—an extremely complex work flow, also involving human intervention.

In our experiments we identify a number of strong inconsistencies, demonstrating that the problem is structural, and not just in syntactic differences. In addition, we show that inconsistencies are not equally spread across different antiviruses (i.e., some vendors are more consistent, while others are wildly different). Also, there is a non-negligible amount of large groups of samples that are labeled inconsistently.

In summary, we make the following contributions:

- We define a systematic technique to create simple yet effective graph-based models of vendors’ naming conventions (§3.2) by means of which we formally define the concept of *consistency*, *weak inconsistency* and *strong inconsistency* (§3.3.2).
- We propose two quantitative measures to evaluate the overall degree of inconsistency between two vendors’ naming conventions (§3.3.2) and, more importantly, we define a simple algorithm to find and extract the portions of graph model that exhibit inconsistencies. The limitations of our technique are clearly described in §5 to help future efforts to extend and refine our approach.
- We describe the results obtained by applying the proposed techniques on a real-world dataset comprising 98,798 unique malware samples, scanned with four real antivirus products, visualize and analyze consistencies, strong and weak inconsistencies, and briefly discuss how these can be solved (§4).

## 2 Malware naming inconsistencies

In the past, although variants of viruses and worms were relatively common, they tended to be just a few children, or a small family tree of descendants. Therefore, even with different conventions (e.g. calling a child “virus.A” as opposed to “virus.1”), such trees were easy to match to each other across different vendors. Even polymorphic viruses did not pose a serious challenge in this scheme. An automated software, VGreep, could be used to perform such mapping [7]. An effort to standardize names was CARO [3], which proposed the following naming convention: `<malware type>://<platform>/<family name>.<group name>.<infective length>.<sub variant><devolution><modifiers>`. However, this effort was unsuccessful. But even if it had been, a standard syntax would solve just a subset of the problem, without attempting to reconcile different family or group names among different vendors.

Polymorphic viral engines gave way to sophisticated packing and obfuscation techniques, and recompilation. This created in turn a rising number of unique malware specimens, which were often slightly mutated versions of known malware. The CME initiative<sup>2</sup> tried to deal with this problem by associating a set of different specimens to a single threat, but the approach proved to be unfeasible. At the same time, most malware authors began to use “malware kits”, and to borrow or steal code from each other. As a

<sup>2</sup> <http://cme.mitre.org/cme/>

result, many samples may descend from a mixture of ancestors, creating complex phylogenies that are certainly not trees anymore, but rather lattices. This, in turn, motivated the evolution of antivirus engines, which now rely on generic signatures including behavior-based techniques inspired by anomaly detection approaches. While this let signature-based scanners survive the rising wave of different malware variants, it also made the task of correlating the outcomes of different antiviruses even more complex [8].

For research and classification purposes however, naming inconsistencies lead to issues when trying to correlate or mine useful data across different antiviruses. For instance, in [2] signature-based antiviruses are compared with behavioral classifiers by means of consistency (i.e., similar samples must have similar labels), completeness and conciseness of the resulting detection. This work has highlighted the presence of a non-negligible number of inconsistencies (i.e., different labels assigned to similar samples).

From the above overview, we conclude that, before attempting to consolidate malware names, a systematic way to quantify, spot consistencies and inconsistencies between malware samples labeled by different vendors is needed.

### 3 Finding naming inconsistency

We hereby describe a two-phase, practical approach to build a high-level picture of inconsistencies in malware naming conventions across a given set of antivirus products or vendors (referred to “vendors” for simplicity). Our goal is to spot inconsistencies that go beyond well-known syntactic differences in malware names. Given a list of unique malware samples (e.g., MD5s, SHA-1), our approach produces a graphical, *qualitative* comparison of a set of *quantitative* indicators—which evaluates the degree of (in)consistency between naming conventions—along with the actual subsets of samples labeled inconsistently.

**Phase 1 (modeling)** For each vendor, we group (or cluster) malware names according to structural similarity between the strings that encode the names (§3.2).



**Phase 2 (analysis)** We compare the aforesaid models quantitatively by means of a set of structural and numerical indicators (§3.3).


For instance, considering samples of backdoor malware, if patterns such as “-backdoor”, “.backdoor.”, “-backdoordialer”, or “backdoor.dialer” are found in their names, we assume that, according to the vendor under examination, they are all characterized by being “backdoors”, and thus **Phase 1** clusters them in the same group. In other words, vendors are modeled by means of the groupings induced by naming conventions. These models are instantiated for each vendor. In **Phase 2**, two vendors are considered consistent if they both group samples together in the same manner, regardless of the actual labeling. For instance, a group comprising sample  $m_1$  (labeled as “foo-backdoor”) and sample  $m_2$  (labeled as “bar-backdoor”) is consistent with a group comprising the same exact samples labeled as “blah-trojan” and “foobar-trojan”, respectively.

We purposely designed **Phase 2** to be agnostic with respect to the technique used to group malware labels (or samples). Thus, it can be used to evaluate the consistency between clusters obtained by other malware clustering tools or constructed manually.

#### 3.1 Types of inconsistency

We focus on two types of inconsistency:

**Weak inconsistency:** One vendor divides the set of samples into more groups, whereas the other vendor groups them all together, therefore creating a “one-to-many” mapping  as opposed to one or more “one-to-one” mappings . This inconsistency is weak as it descends from the different granularities adopted by vendors).

**Strong inconsistency:** Vendors spread samples in multiple groups, such that there is no mapping between the groups , i.e. such that it is impossible to reduce all the inconsistencies to weak inconsistencies.

In §3.3.2 we further formalize these cases by means of models constructed in **Phase 1** and define a fast algorithm to spot them.

### 3.2 Phase 1: Naming convention modeling

We model the structural characteristics of naming conventions by grouping malware samples based on their *label*, i.e., the string that encodes the malware name. We adopt a simplified, top-down hierarchical clustering approach (§3.2.2), which recursively splits an initial set (or cluster) of malware samples into nested sub-clusters. The procedure ensures that each (sub-)cluster contains only samples labeled with similar *string patterns*. Patterns are extracted offline for each vendor (§3.2.1).

**3.2.1 Pattern extraction** Our technique is centered around four *pattern classes*, marked from hereinafter between angular brackets in teletype font (e.g., <class>):

- threat <type> indicates a distinctive activity performed by the malicious code. For example, this class of string patterns captures substrings as “backdoor”, “worm”, or “dialer”, “packed”, “tool”.
- <family> indicates the name of a specific malware family (e.g., “Conficker”, “Mudrop”, “Fokin”, “Allapple”).
- <platform> indicates the (target) platform, operating system (e.g., “W32”, “WNT”) or interpreter (e.g., “JS”, “PHP”), required to execute the malware and possibly infect the victim.
- <version> indicates the version of the malicious code (e.g., “B” and “D” in labels “PHP:IRCBot-B” and “PHP:IRCBot-D”), or additional information to disambiguate various “releases” or signature (e.g., “gen”, “gen44”, “damaged”).

This small, generic set of pattern classes allows to analyze several vendors. However, new classes can be easily added to extend our approach to virtually any vendor. Based on our analysis on real malware samples, each class can either contain one simple pattern or a hierarchy of simple patterns:

- A *simple pattern* is the occurrence of a string of a given class, e.g., <type> = Trojan. Classes of patterns such as the target platform and the malware family usually occur as simple patterns (e.g., <platform> = Win32|PHP).
- A *hierarchy of simple patterns* is the occurrence of more simple patterns of the same class, e.g., <type1> = “Trojan” and <type2> = “Dropper” are both of class <type>. For example, when vendors specify both a threat type and sub-type, this leads to hierarchies of simple patterns, denoted as concatenated simple patterns in order of precedence, e.g., <type> = <type1>/<type2>/<type3>, <version> = <version1>/<version2>. Note that, the slash separator is simply a convention used to describe our results and by no means reflects the order of occurrence of the patterns.

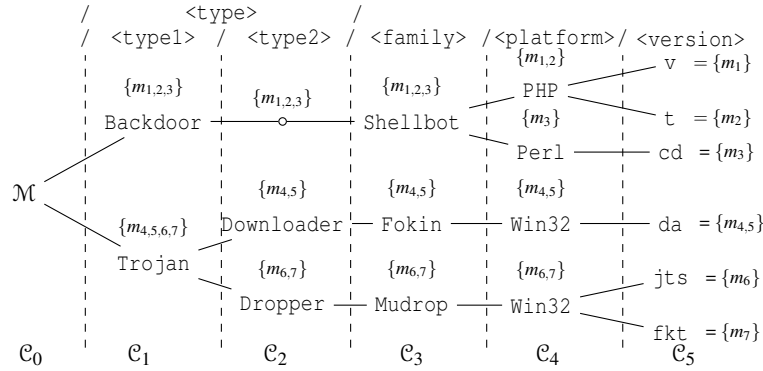


Figure 1: Example output of **Phase 1** on a set  $\mathcal{M}$  with seven samples. For instance,  $\mathcal{C}_1 = \{\{m_1, m_2, m_3\}, \{m_4, m_5, m_6, m_7\}\}$  and  $\mathcal{C}_{2-4} = \{\{m_3\}, \{m_6, m_7\}, \{m_4, m_5\}, \{m_1, m_2, m_3\}, \{m_1, m_2\}\}$ . Note that,  $\langle \text{type} \rangle$  comprises  $\langle \text{type1} \rangle / \langle \text{type2} \rangle /$ .

Simple patterns can be constructed either manually from a handful of labels, or by leveraging automatic inference tools to derive the most probable syntax of a given set of strings for subsequent manual revision. However, since manual revision would be required anyway to ensure accurate results, we opt for a heuristic approach (detailed in §3.2.3), that allows us to extract the patterns in a semi-automatic fashion. Hierarchies of patterns of the same class are ordered with respect to their relative frequency of appearance. For instance, given one vendor and simple patterns  $\langle \text{typeX} \rangle$  and  $\langle \text{typeY} \rangle$ ,  $X < Y$  if  $\langle \text{typeX} \rangle$  occurs more than  $\langle \text{typeY} \rangle$  on a given set of malware sample. If they have the same frequency, the hierarchy is replaced by a simple pattern  $\langle \text{type} \rangle$ , which contain the common substring between  $\langle \text{typeX} \rangle$  and  $\langle \text{typeY} \rangle$ .

**3.2.2 Clustering** The clustering is run for each vendor given a set of patterns and a set  $\mathcal{M}$  of malware samples (and the corresponding labels). The algorithm proceeds by considering one class of patterns at time and is best described by means of an example.

A first split is obtained according to the pattern class  $\langle \text{type} \rangle$ . In a simple example, given the substrings “Backdoor” and “Trojan” of pattern  $\langle \text{type} \rangle$ , the samples labeled as Backdoor.Pperl.Shellbot.cd, Backdoor.PHP.Shellbot.v and Backdoor.PHP.Shellbot.t fall in the same cluster, while Trojan-Downloader.Win32.Fokin.da, Trojan-Dropper.Win32.Mudrop.fkt and Trojan-Dropper.Win32.Mudrop.jts fall in a different one. If the vendor under consideration adopts hierarchical patterns, this step is repeated for each sub-pattern. Continuing the above example, the trojan samples are separated in two different clusters.

When a cluster can be split no further according to the same pattern class, the procedure considers the malware family. In our example, the only possible split is by means of “Fokin” and “Mudrop”, as “Shellbot” induces no splits. Then the first cluster is split in two sub-clusters, one containing only Backdoor.Pperl.Shellbot.cd and one with Backdoor.PHP.Shellbot.t Backdoor.PHP.Shellbot.v. Further splits are performed according to the different  $\langle \text{version} \rangle$  patterns (if any). More precisely, “.v” and “.t” forms two sub-clusters as well as “.fkt”, and “.jts” do.

The procedure stops when all possible splits have been made, i.e., when the latest pattern class has been considered. In our example, the procedure ends after one split in-

duced by the version. At each split, relationships between super-clusters and sub-clusters are stored to construct a cluster tree, rooted in the initial set. The output for the above example is depicted in Fig. 1.

**Definition 1 (Cluster tree).** *Given a set  $\mathcal{M}$  of malware names, we define the output of Phase 1 as  $\mathcal{C}_d(\mathcal{M}) \subset \wp(\mathcal{M})$  called cluster tree, where  $d$  is either: (1) a number that indicates the depth in the tree, e.g.,  $\mathcal{C}_1$ , (2) an interval between depths in the tree, e.g.,  $\mathcal{C}_{1-2}$ , or (3) a mnemonic expression ( $\mathcal{M}$  is omitted in the following when implicit from the context).*

Considering the example tree in Fig. 1,  $\mathcal{C}_1 = \{\{m_1, m_2, m_3\}, \{m_4, m_5, m_6, m_7\}\}$  and  $\mathcal{C}_{2-4} = \{\{m_3\}, \{m_6, m_7\}, \{m_4, m_5\}, \{m_1, m_2, m_3\}, \{m_1, m_2\}\}$ . The whole tree is  $\mathcal{C} = \mathcal{C}_0 = \mathcal{C}_0(\mathcal{M}) = \{\mathcal{M}\}$ , or  $\mathcal{C}^v$ , where  $v$  is the vendor under examination. Considering the example tree in Fig. 1, clusters at depth 3 can be indicated by means of mnemonic expression “/\*/<family>/\*”, which denotes samples in  $\mathcal{M}$  clustered according to their family, i.e.,  $\mathcal{C}_3 = \mathcal{C}_{(/*/<family>/*)} = \{\{m_1, m_2, m_3\}, \{m_4, m_5\}, \{m_6, m_7\}\}$ . Actual substrings can be used as well, e.g., /Backdoor/\* is  $\mathcal{C}_{/Backdoor/*} = \{\{m_1, m_2, m_3\}\}$ . Note that, a hierarchy of patterns always indicate clusters at the lowest depth. For instance,  $\mathcal{C}_2 = \mathcal{C}_{(/*/<type2>/*)} = \{\{m_1, m_2, m_3\}, \{m_4, m_5\}, \{m_6, m_7\}\}$ .

### 3.2.3 Implementation details

*Pattern extraction:* The heuristic extraction procedure is run for each vendor and takes (1) a set of malware labels  $\mathcal{L}$  and (2) an a small set of *separators*,  $[/:.-_!]$  (this can be customized easily by analyzing the frequency of symbols in the labels corpus). The algorithm is semi-supervised and proceeds by iteratively breaking labels into substrings. At each iteration an operator reviews a set of candidate substrings and assign them to one appropriate pattern class. Pattern classes are initially empty, e.g.,  $\langle \text{type} \rangle = ''$ . At the  $i$ -th iteration a random, small (e.g., 10) subset of labels  $\mathcal{L}_i \subseteq \mathcal{L}$  is selected and labels are broke into substrings according to separators. Then, the operator assigns each unique substring to the appropriate class. For example, if Win32, Allapple, Trojan, and PHP are found, the appropriate class is updated, i.e.,  $\langle \text{platform} \rangle_i = \text{Win32|PHP}$ ,  $\langle \text{type} \rangle_i = \text{Trojan}$ ,  $\langle \text{family} \rangle_i = \text{Allapple}$ . All substrings extracted from each label in  $\mathcal{L}_i$  must be assigned to exactly one class. Labels with at least one substring not assigned to any class are postponed for subsequent analysis (and removed from  $\mathcal{L}_i$ ). Alternatively, the operator can add new separators as needed to handle the current subset of labels. When labels in  $\mathcal{L}_i$  are covered,  $\mathcal{L}$  is reduced by removing all the labels that can be parsed with the existing patterns. Then, the next random sample  $\mathcal{L}_{i+1} \subseteq \mathcal{L} \setminus \mathcal{L}_i$  is drawn. Note that,  $\mathcal{L}_{i+1}$  may include postponed labels. This procedure continues until  $\mathcal{L} = \emptyset$ .

The larger each random sample size is, the faster and more accurate this procedure becomes, also depending on the operator’s experience. However, this procedure needs to be ran only once per vendor and, more importantly, the time and effort required decrease from vendor to vendor, since patterns can be reused (e.g., family and platforms recur across vendors with minimal variations). In real-world examples, a minority of labels may deviate from the patterns (e.g. when labels are handwritten by malware analysts). In our experiments, we avoid to create clusters not reflecting the actual semantics of a vendor’s naming convention by manually inspecting the (small) sets of outliers that match no patterns.

*Cluster splitting:* We found clusters with only one sample, i.e., singletons. For example, consider patterns  $\langle \text{version} \rangle = v|t$  and a cluster  $\{m_1, m_2\}$ , where  $m_1 =$

`Backdoor.PHP.Shellbot.v`,  $m_2 = \text{Backdoor.PHP.Shellbot.t}$ . A split would produce two subclusters  $\{m_1\}, \{m_2\}$ .

To one end, one outlier is not representative of the pattern, e.g., “t” or “v”. To the other hand, since our goal is to analyse consistency, we expect that, if two vendors are consistent, they would produce similar clusters, also including “outliers”. For this reason, to take into account both the observations, clusters of size below a certain threshold,  $T_o$ , are labeled with a special pattern, `<misc>` that encode such “uncertainty”.

For example, while `/Backdoor/Shellbot/PHP/` identifies the set  $\{m_1, m_2\}$ , the label `/Backdoor/Shellbot/<misc>/` identifies  $\{m_3\}$ . Note that, more miscellaneous clusters may exist at the same depth.

*Depth:* We use the mnemonic expressions to indicate the depth in a cluster tree. However, for different vendors, a named depth, e.g., “<family>”, may correspond to different numerical depths. In our implementation, we keep track of the depth in each tree to allow such queries without knowing the actual numerical depth.

### 3.3 Phase 2: Comparing vendors

In this phase cluster trees defined in §3.2 are leveraged as models that characterize the structural properties of the naming convention used by a given vendor. Hence, two vendors  $A, B$  are compared by means of their cluster trees  $\mathcal{C}^A, \mathcal{C}^B$  (Def. 1). First, two numerical indicators are calculated as described in §3.3.1 to quantify the degree of inconsistency between naming conventions between vendors. Secondly, a technique to extract inconsistencies is described in §3.3.2.

Cluster trees are hierarchies of sets (see Fig. 1). However, the following analysis compares sets, derived by “cutting” cluster trees at a given depth  $d$ , which is omitted to simplify the notation. In other words, from hereinafter  $\mathcal{C}^A = \mathcal{C}_d^A$  and  $\mathcal{C}^B = \mathcal{C}_d^B$ .

**3.3.1 Quantitative comparison** In this section, two indicators of inconsistency are defined. The (1) naming convention distance  $D(\mathcal{C}^A, \mathcal{C}^B) \in [0, 1]$  expresses the overall difference between the naming conventions adopted by  $A$  and  $B$ , while (2) the scatter measure  $S(\mathcal{C}^A, \mathcal{C}^B)$  expresses the average number of clusters of one vendor that are necessary to cover each cluster of the other vendor (and vice-versa).

**Definition 2 (Naming convention distance).** *The naming convention distance between vendors  $A$  and  $B$  is defined as the average distance between their clusters.*

$$D(\mathcal{C}^A, \mathcal{C}^B) := \frac{1}{2} \left( \frac{\sum_{c \in \mathcal{C}^A} \delta(c, \mathcal{C}^B)}{|\mathcal{C}^A|} + \frac{\sum_{c \in \mathcal{C}^B} \delta(c, \mathcal{C}^A)}{|\mathcal{C}^B|} \right) \quad (1)$$

$\delta(c, \mathcal{C}') = \min_{c' \in \mathcal{C}'} d(c, c')$  being the minimum diff. between  $c \in \mathcal{C}$  and any set of  $\mathcal{C}'$ .

The denominator is a normalization factor to ensure that  $D(\cdot, \cdot)$  is in  $[0, 1]$ . A similar distance has been proposed in [12] to measure the similarity between sets of overlapping clusters, although we propose a different normalization factor.

The difference between sets  $d(c, c')$  can be computed in several ways as our methodology only requires it to be in  $[0, 1]$ . In our study, we choose two variants:  $d_J(c, c') = 1 - J(c, c')$  and  $d_H(c, c') = \frac{H(X_c|X_{c'})}{H(X_c)}$ , where  $J(c, c')$  is the Jaccard index,  $H(X_c|X_{c'})$  is the conditional entropy, and  $H(X_c)$  is the entropy of a random variable [11, p.138]. Both

$X_c$  and  $X_{c'}$  are random variables that encode sets  $c$  and  $c'$ , respectively, by means of a binomial process, i.e.,  $X_c \sim \text{Bin}(\frac{|c|}{|\mathcal{M}|})$ , and  $X_{c'} \sim \text{Bin}(\frac{|c'|}{|\mathcal{M}|})$ . A similar distance function have been used in [10] to calculate the degree of overlapping in graphs representing hierarchical communities. As discussed in §4, although the values of  $d_J$  and  $d_H$  differ, the results are not influenced.

**Definition 3 (Scatter measure).** *The scatter measure between vendors  $A$  and  $B$  is defined as the average number of sets in each vendor's model that are necessary to cover one set drawn from the other vendor's model (and vice-versa). More formally:*

$$S(\mathcal{C}^A, \mathcal{C}^B) := \frac{1}{2} \left( \frac{\sum_{c \in \mathcal{C}^A} |\Gamma(c, \mathcal{C}^B)|}{|\mathcal{C}^A|} + \frac{\sum_{c \in \mathcal{C}^B} |\Gamma(c, \mathcal{C}^A)|}{|\mathcal{C}^B|} \right) \quad (2)$$

where  $\Gamma(c, \mathcal{C}')$  is the scatter set.

**Definition 4 (Scatter set).** *The scatter set of  $c$  with respect to  $\mathcal{C}'$  is:*

$$\Gamma(c, \mathcal{C}') := \{c' \in \mathcal{C}' \mid c \cap c' \neq \emptyset\}. \quad (3)$$

In other words,  $\Gamma$  contains sets of  $\mathcal{C}'$  (e.g., model of vendor  $B$ ) that have at least one element (e.g., malware sample) in common with a given  $c \in \mathcal{C}$  (e.g., model of vendor  $A$ ). Note that, since  $\mathcal{C}^A$  and  $\mathcal{C}^B$  are *partitioned*,  $|\Gamma(c, \mathcal{C}')|$  is the no. of sets of  $\mathcal{C}'$  that build  $c$ .

**3.3.2 Structural comparison** In this section, a method to recognize inconsistencies (as intuitively defined in §3.1) is defined. To this end, clusters in a tree at a given depth are first represented as undirected graphs with cross-vendor edges, and then searched for inconsistent sub-graphs. Note that, this comparison can be made only if  $\mathcal{C}^A$  and  $\mathcal{C}^B$  are partitioned into flat clusters (i.e., sets). For this reason, only for this analysis, from hereinafter (and in §4.3.3) we assume that clusters are drawn from trees at leaf depth (i.e., `<version.n>`), representative of the whole label.

**Definition 5 (Graph of cluster trees).** *Given  $\mathcal{C}^A$  and  $\mathcal{C}^B$  the graph of cluster trees is  $\mathcal{G}^{AB} := \langle \mathcal{V}^{AB}, \mathcal{E}^{AB} \rangle$ , where  $\mathcal{V} = \mathcal{C}^A \cup \mathcal{C}^B$  and  $\mathcal{E}^{AB} = \{(c, c') \mid c \in \mathcal{C}^A, c' \in \mathcal{C}^B \wedge c \cap c' \neq \emptyset\}$ .*

In other words,  $\mathcal{G}^{AB}$  represents the relations existing between sets of labeled malware samples. Given a set  $c$  containing samples labeled by  $A$ , and a set  $c'$  containing samples labeled by  $B$ , an edge from  $c$  to  $c'$  is created only if  $c'$  has at least one sample in common with  $c$ . In §3.3.3 this concept is extended and edges are weighted proportionally to the number of samples shared between  $c$  and  $c'$ . Therefore, the problem of recognizing inconsistencies between cluster trees (i.e., between vendors) consists in finding connected components of  $\mathcal{G}^{AB}$ , for which efficient algorithms (e.g., [13]) are already implemented in common programming language library functions. Each connected component  $\mathcal{G}_c^{AB} \subset \mathcal{G}^{AB}$  is then analyzed automatically to distinguish among:

**Consistency (CC)** (Fig. 2a) The connected component is made of two clusters containing identical malware samples. There is a consistency even when samples of  $A$  have different labels than samples of  $B$ .



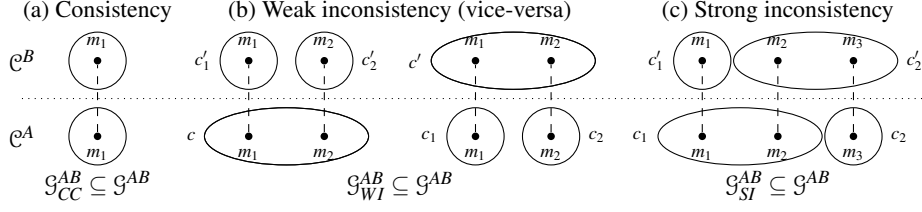


Figure 2: Instances of consistencies  $\mathcal{G}_{CC}^{AB}$ , weak inconsistencies  $\mathcal{G}_{WI}^{AB}$  and strong inconsistencies  $\mathcal{G}_{SI}^{AB}$ , i.e., connected components of graph models  $\mathcal{G}^{AB}$  of vendors  $A$  vs.  $B$ . Each vertical line represents a malware sample.

**Weak Inconsistency (WI)** (Fig. 2b) The connected component contains only one cluster  $c \in \mathcal{V}^A = \mathcal{C}^A$ , and all clusters  $c' \in \mathcal{V}^B = \mathcal{C}^B$  are its subsets  $c' \subset c$ . In this case, vendor  $B$  adopts more fine-grained naming convention than vendor  $A$ . Despite  $\mathcal{C}^A$  and  $\mathcal{C}^B$  are not identical, vendors disagree only on the amount of information encoded in each label.

**Strong Inconsistency (SI)** The connected component contains more than one cluster for each vendor (e.g., for clusters  $c'_1, c_1, c'_2, c_2$  in Fig. 2c). Since clusters are partitions of the entire set of malware samples, there must be at least four clusters  $c_1, c_2 \in \mathcal{V}^A = \mathcal{C}^A$ ,  $c'_1, c'_2 \in \mathcal{V}^B = \mathcal{C}^B$  such that the following condition holds:

$$c_1 \cap c'_1 \neq \emptyset \wedge c_2 \cap c'_2 \neq \emptyset \wedge c'_2 \cap c_1 \neq \emptyset$$

The inconsistency includes all clusters of the connected component. In other words, the clusters share some samples without being all subsets of each other. This is caused by inherently different naming conventions. Once found, these inconsistencies can be solved by fusing, say,  $c_1$  with  $c_2$ .

### 3.3.3 Implementation details

*Accurate distance between sets:* In our implementation, the distance function  $d_H(c, c')$  mentioned in §3.3.1 is defined as:

$$d_H(c, c') = \begin{cases} \frac{H(X_c|X_{c'})}{H(X_c)} & \begin{aligned} &h(P(1, 1)) + h(P(0, 0)) > h(P(1, 0)) + h(P(0, 1)) \text{ (a)} \\ &\vee \frac{H(X_c|X_{c'})}{H(X_c)} - J(c, c') < T_H \text{ (b)} \end{aligned} \\ 1 & \text{otherwise} \end{cases} \quad (4)$$

where the probabilities  $P(\alpha, \beta) = P(X_c = \alpha, X_{c'} = \beta)$  are derived from the marginals  $P(X_c = \alpha), P(X_{c'} = \beta)$ , estimated by modeling the membership of an element in its set as a binomial process (as described in §3.3.1).  $h(\cdot)$  is the entropy. The original condition (a) avoids  $d_H(c, c') = 0$  when  $X_c = f(X_{c'})$  (e.g.,  $c \cap c' = \emptyset$  and  $c \cup c' = \mathcal{M}$ ). Based on our experiments, although such distance works well in general, it may lead to biased results. For example, consider  $|c| = |c'| = 10$ ,  $|c \cap c'| = 4$ , and a realistic dataset comprising 98,798 unique malware samples. Although  $c$  and  $c'$  have a 40% overlap, with the original condition (a)  $d_H(c, c') = 1$ . For this reason, we relax (a) by introducing (b), which avoids penalizing small clusters that have non-negligible overlapping. Our proposed distance indeed accounts for the degree of overlapping with the Jaccard index  $J(c, c')$ . In the aforesaid example, the modified distance would be about 0.69 (more reasonable than 1).

VENDOR	SYNTAX	#LABELS
Microsoft $V_1$	$\langle \text{type} \rangle : \langle \text{platform} \rangle / \langle \text{family} \rangle [ \langle \text{gen} [! \langle \text{version1} \rangle]   \langle \text{version2} \rangle ]$	4,654
Antiy $V_2$	$\langle \text{type} \rangle . \langle \text{platform} \rangle / \langle \text{family} \rangle [ . \langle \text{version1} \rangle . \langle \text{version2} \rangle ]$	23,603
Kaspersky $V_3$	$\langle \text{type} \rangle / \langle \text{platform} \rangle . \langle \text{family} \rangle [ \langle \text{gen} \rangle ]$	2,122
Avast $V_4$	$\langle \text{platform} \rangle / \langle \text{family} \rangle [ - \text{gen}   - \langle \text{version} \rangle ]$	4,350

Table 1: Vendors analyzed in our experiments.

The threshold  $T_H$  can be derived fairly easily from the dataset under consideration, which, in our case, contains a majority of small clusters (i.e., between 1 and 100). More precisely, for every possible size  $|c|, |c'| \in [1, 100]$ , and for every possible intersection  $|c \cap c'|$ , we computed  $|d_H - d_J|$  and noticed that its distribution is centered around low values. Thus, we set  $T_H$  to 0.1, as  $|d_H - d_J| < 0.1$  in 80%. Low values of  $T_H$ , i.e.,  $T_H \simeq 0.1$ , tend to produce more realistic comparisons between vendors, because the distance function is smoother and the extreme case  $d_H = 1$  occurs when sets are actually very different, e.g., when  $c \cap c' = \emptyset$ .

*Scatter set coverage:* Our implementation incorporates a measure of *coverage*,  $\sigma$ , in scatter sets  $\Gamma(c, \mathcal{C}')$  (Def. 4), defined as:

$$\sigma(\Gamma) := \frac{\left| c \cap \left( \bigcup_{c' \in \Gamma(c, \mathcal{C}')} c' \right) \right|}{|c|} \%, \quad (5)$$

which quantifies the percentage of samples in  $c$  (e.g., a cluster of vendor  $A$ ) shared with the union of scatter sets derived from  $\mathcal{C}'$  (e.g., a cluster tree of vendor  $B$ ). Scatter sets can be selected with respect to their  $\sigma$ , and thus, given a threshold  $T_\sigma \in [0, 100]$ , the *minimum scatter set* of  $c$  with respect to  $\mathcal{C}'$  can be selected as  $\hat{\Gamma}_{T_\sigma} : \# \Gamma(c, \mathcal{C}') \text{ for } \sigma(\Gamma) \geq T_\sigma \wedge |\Gamma| < |\hat{\Gamma}|$ , which is the smallest scatter set that covers  $c$  of at least  $T_\sigma$ .

*Weighted structural models:* The edges of graphs of cluster trees (Def. 5) are weighted with the following weighting function:

$$W(c, c') := \max \left\{ \frac{|c \cap c'|}{|c|} \%, \frac{|c \cap c'|}{|c'|} \% \right\}, \forall (c, c') \in \mathcal{E}^{AB} \quad (6)$$

Each edge encodes the degree of “overlapping” between two clusters  $c$  and  $c'$  originated from  $A$  and  $B$ , respectively. Note that, our normalization ensures that weights quantify the actual fraction of  $c$  shared with  $c'$ , regardless of the size of  $c'$ , which can be proportionally larger than  $c$  (and vice-versa). Our analysis can be thus parametrized by a threshold  $T_W \in [0, 100]$ , used to convert weighted graphs into graphs by pruning edges  $e = (c, c')$  below  $T_W$ , i.e.,  $W(c, c') < T_W$ .

## 4 Experimental measurements

We focus on the four vendors listed in Table 1, whose conventions cover the vast majority of the samples and are derived from the analysis of the labels contained in the dataset. These vendors are good candidates, first because the “richness” of their naming convention allow a granular analysis, that spans from  $\langle \text{type} \rangle$  to  $\langle \text{version} \rangle$ , and secondly because the regularity of their labels allowed us to extract patterns with reasonable efforts as discussed in §5. Adding more vendors is computationally feasible

and our method does not prevent this. However, the number of unique couples drawn from the set of vendors would grow quickly. Therefore, from a presentation perspective, this may yield cluttered and confusing diagrams. Given that the goal of the evaluation is showing that our method finds structural inconsistencies—and not only to quantify them, we argue that four vendors, totaling six comparisons, are sufficient.

Vendor  $V_4$  includes no `<type>` pattern class. We manually analyzed this case and discovered that the `<family>` pattern class, which is instead present in the syntax, is very seldom used to hold information about the threat type (e.g., “Malware”, “Dropper”, “Trojan” in Fig. 4). Since this phenomenon is very limited, it is reasonable to consider it as part of the semantic of the naming convention. For this reason, only for vendor  $V_4$ , we safely consider threat type and family name at the same level of importance. Note that, other vendors “circumvent” this minor issue by assigning `<family>` = “generic”.

*Dataset:* Our dataset  $\mathcal{M}$ , generated on September 13, 2010, comprises 98,798 unique malware samples, identified by their hashes, and labels  $\mathcal{L}^{V_1}, \mathcal{L}^{V_2}, \mathcal{L}^{V_3}, \mathcal{L}^{V_4}$  have been derived with VirusTotal, an online service to scan samples with multiple vendors simultaneously. More precisely, we first queried VirusTotal for the MD5s of the top 100,000 submitted binaries. Second, we selected four vendors, which gave us with a subset of 98,798 unique MD5s recognized by all of them. Frequent labels in the datasets include, for instance, “TrojanSpy:Win32/Mafod!rts”, “Net-Worm.Win32.Allapple.-b”, “Trojan/Win32.Agent.gen”, “Trojan:Win32/Meredrop”, “Virus.Win32.Induc.-a”. A minority of labels deviates from these conventions. For example, in  $V_4$  only eight labels (0.00809% of the dataset) contain “gen44” instead of “gen”. Similarly, five labels (0.00506% of the dataset) of  $V_2$  contain a third version string. Other cases like the “@mm” suffix in  $V_1$  labels (101 labels i.e., 0.10223% of the dataset) do not fit in the above convention. Note that, from a purely syntactic point of view, these cases are similar to the presence of keywords often used by analyzers to mark special samples, e.g., “packed”. We handled these handful of outliers.

**Phase 1** was run on the dataset to create a model, i.e., cluster tree, for each vendor,  $\mathcal{C}^{V_1}, \mathcal{C}^{V_2}, \mathcal{C}^{V_3}, \mathcal{C}^{V_4}$ . Next, quantitative and structural analysis of **Phase 2** have been run. The outcome of this experimentation is presented and discussed in the remainder of this section, after a brief visual overview of the cluster trees.

#### 4.1 Cluster tree visual comparison

Differences between naming conventions induce cluster trees that, at a high level, are structurally dissimilar. As shown in Fig. 3, even at a first glance, each vendor group malware samples differently. For example,  $V_4$  tends to form clusters very early by splitting samples in a large number of clusters based on `<family>`. Contrarily, other vendors in our dataset form sparser cluster trees, revealing the adoption of sophisticated (or simply, more fine-grained) naming conventions.

A concrete example can be seen in the magnified portions of cluster trees shown in Fig. 4. For ease of visualization, we extracted 50 samples all falling in the same cluster, i.e., all classified as `packed-Klone` by  $V_3$ , used as a comparison baseline. Vendor  $V_1$  organizes the same set of samples onto 8 clusters, including `worm`, `trojan`, and `pws` subsets. Also, a separate subset holds samples not even considered malicious by  $V_1$ . This example, drawn from a real dataset, also shows that, as expected, the labels’ granularity varies across vendors. For instance,  $V_1$ , which adopts a granularity ranging from 1 to 3, splits `worms` (depth 1) in `Pushbots` and `Pushbot` (depth 2). This can be seen by noticing

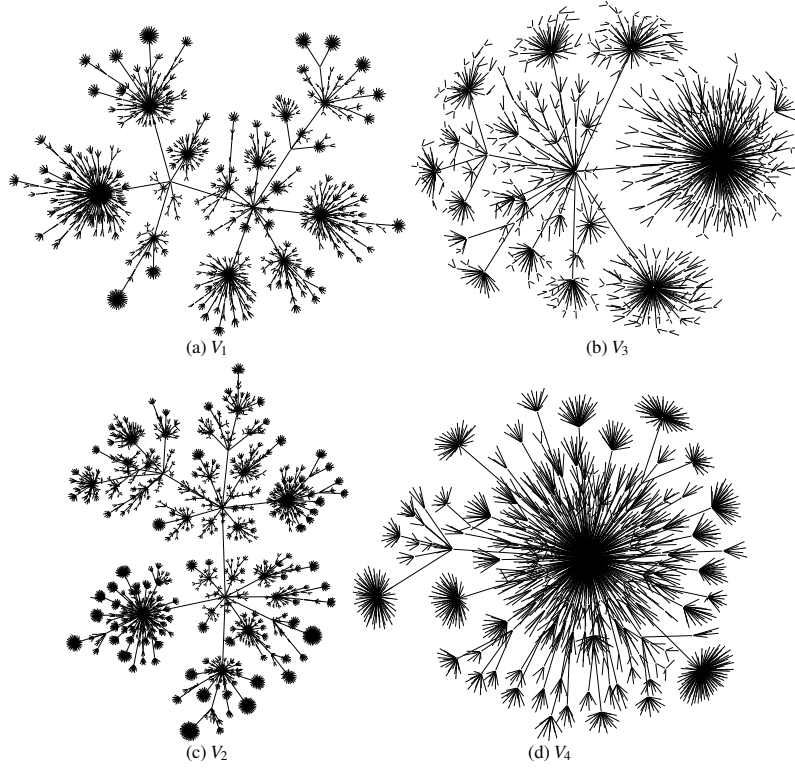


Figure 3: Visual comparison of cluster trees of each vendor.

the dash “-” separator. However, the splitting is not visible in this small set, because (as described in §3.2.3), we avoid creating singleton clusters, and `Pushbot` (for instance) would indeed contain only  $m_{38}$ . Instances of the same behavior are visible in  $V_2$ , which labels’ depth ranges in 1-4, and  $V_4$ , which labels’ depth is 1 (according to this excerpt). We now analyze these “visual” discrepancies thoroughly.

#### 4.2 Singleton clusters and “not detected” samples

Depending on the value of  $T_W$ , we observed the creation of isolated singleton nodes (i.e., clusters of one vendor with no corresponding cluster in the counterpart). Depending on the structure of the graph, by varying  $T_W$ , these nodes may either link to another single node (thus creating a consistency), or to several nodes (thus creating an inconsistency). An optimistic interpretation, would count them as consistencies because, for a certain, low value of  $T_W$ , at least one corresponding cluster exists. On the other hand, a pessimistic interpretation would consider such nodes as potential inconsistencies. Due to the inherent ambiguity and unpredictability of this phenomenon, we ignore singleton nodes to avoid biased results.

Another peculiar type of nodes are those originated by clusters with samples not detected as malicious by the vendor under examination (labeled as “not detected”, e.g., in Fig. 4), treated by our pattern extraction procedure as a label containing only a `<type_n>` string. These are not regular inconsistencies, since are not caused by actual discrepancies between naming conventions, but they depend on the accuracy of the

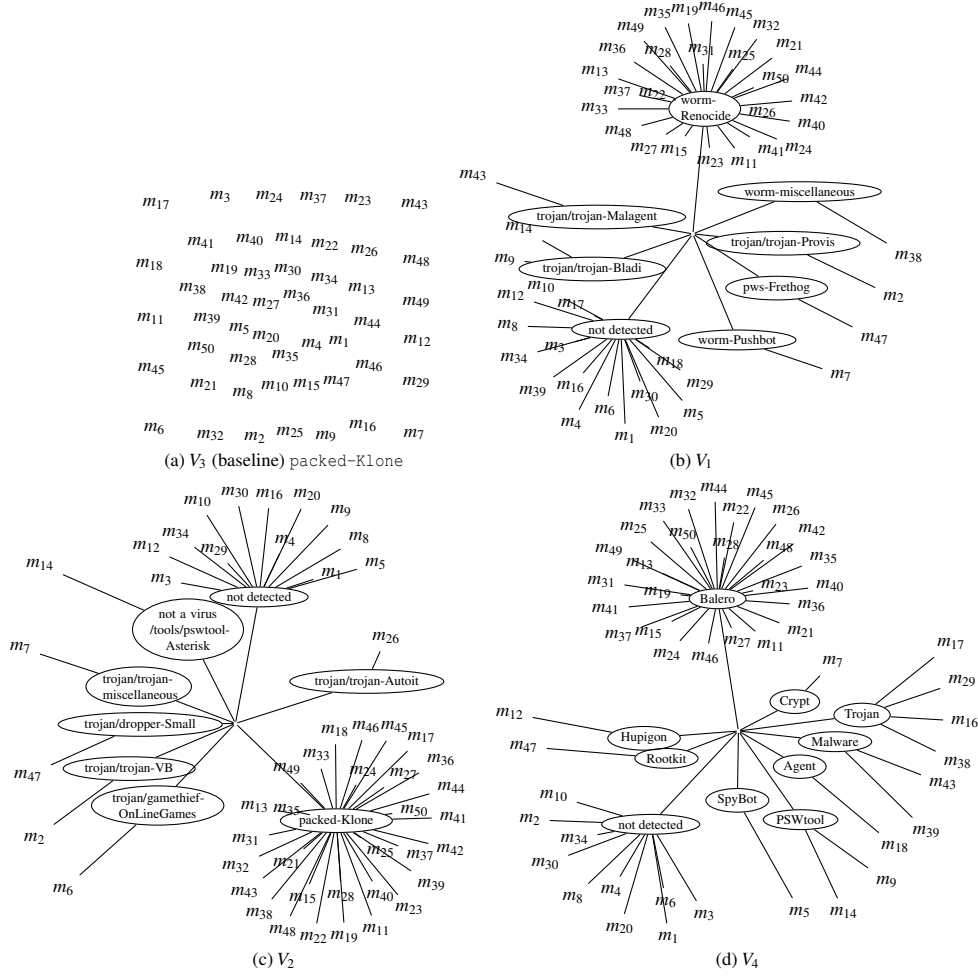


Figure 4: A flat cluster extracted from  $V_3$ 's cluster tree cut at  $\langle \text{family} \rangle$  depth. Other vendors group the same set of samples differently. Only one baseline cluster is shown, although the same behavior can be observed also with other baselines, i.e.,  $V_1$ ,  $V_2$ ,  $V_4$ .

detection. Indeed, they can originate from false positives or false negatives. It is interesting to observe that these nodes are linked to several other clusters, yielding a minority of very large inconsistencies (possibly including the whole graph). This may bias the quantitative comparison. Hence, we removed such clusters from the following analysis. More precisely, the scatter measure discussed in §4.3.2 ignores the scatter sets originating from these clusters. Similarly, the graph of cluster trees (Def. 5) used for structural comparison, discussed in §4.3.3, was pruned by removing “not detected” clusters (i.e., nodes). Also, for consistency with the choice of excluding singleton nodes, we also removed nodes *only* connected to such nodes.

### 4.3 Quantitative comparison

The results of our quantitative measurements are presented and discussed in this section.

**4.3.1 Naming convention distance** Distances between cluster trees are quantitative indicators of the overall inconsistency between the vendors' naming conventions. We calculated the two variants of the naming convention distance,  $D(\mathcal{C}^A, \mathcal{C}^B)$  (defined in §3.3.1), for each unique couple of vendors  $A$  vs.  $B$ , and summarized in Fig. 5. The val-

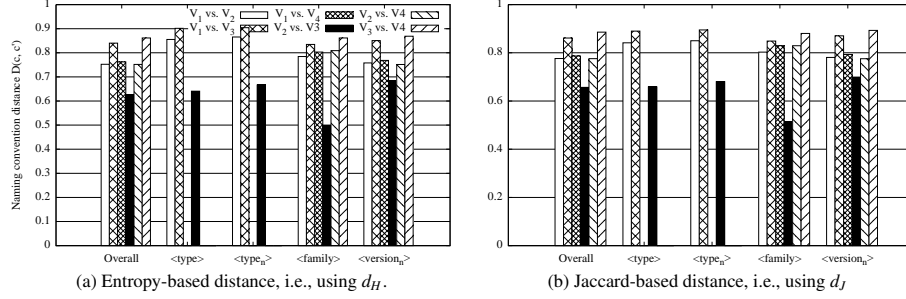


Figure 5: Vendor comparison with two variants of naming convention distance at different depths of the cluster trees. Relatively high distance between vendors is observed from both Entropy- and Jaccard-based distance. Notably, the depth (e.g., `<type>`, `<family>`) negligibly influences the distances, except for  $V_2$  vs  $V_3$ , which exhibit slightly more similarity in terms of `<version_n>`.

ues of both the Entropy-based variant  $d_H$  in Fig. 5a and the Jaccard-based variant  $d_J$  in Fig. 5b lead to the same conclusions. Specifically, it can be observed that the overall consistency is higher (i.e., distance is lower) at `<version_n>` depth than at `<family>` depth, and is also higher at `<family>` than at `<type>` depth. Interestingly, this contradicts the intuitive conjecture that lower levels in the cluster tree would exhibit progressively lower consistency. We also notice that vendors  $V_2$  and  $V_3$  show remarkably more consistent naming convention with respect to the others, especially at `<family>` depth. These two vendors exhibits small structural inconsistencies as also noted in §4.3.3.

**4.3.2 Scatter measure** The scatter measure between cluster trees is a quantitative indicator of the overall degree of “spreading” of clusters from one tree,  $\mathcal{C}^A$ , onto clusters of the other one,  $\mathcal{C}^B$  (averaged with the vice-versa). More precisely, as detailed in §3.3.3, the scatter measure can be calculated at different percentages of coverage,  $\sigma(\Gamma)$ , of the scatter set,  $\Gamma$  (i.e., the set of clusters in  $\mathcal{C}^B$  corresponding to the set  $c \in \mathcal{C}^A$  under examination, and vice-versa). This is done from  $A$  to  $B$  and vice-versa, and by varying a threshold  $T_\sigma$ . Therefore, we calculated  $S(\mathcal{C}^A, \mathcal{C}^B)$  for  $T_\sigma \in \{1\%, 5\%, 10\%, \dots, 95\%, 100\%\}$ : low values of  $T_\sigma$  lead to lower, optimistic, values of  $S(\cdot, \cdot)$ , reflecting the existence of small scatter sets, which are selected albeit they cover only a slight portion of the cluster under examination. Contrarily, higher values of  $T_\sigma$  would unveil the *real* scatter sets, i.e., those with substantial overlapping.

Fig. 6 summarizes the results of this experiment for each couple of vendors at different “cuts” of the cluster trees, i.e.,  $d \in \{\text{<type_n>, <family>, <version_n>}\}$ . As expected from previous analysis (yet contradicting intuitive presuppositions), the scatter measure decreases at lower depths, except for  $V_2$  vs.  $V_3$ , which reveal once again their overall consistency, especially at `<family>` level. Notably, this confirms the conclusions derived from Fig. 5.

Another interesting comparison is  $V_1$  vs.  $V_3$ , which, according to Fig. 5, show remarkable distance and thus can be considered different from one another. First, Fig. 6 confirms this conclusion. In addition, it is worth noticing that these vendors tend to have divergent scatter measures (for increasing values of  $T_\sigma$ ), especially at `<type_n>` depth (Fig. 6a), thus revealing that vendors disagree more on threat types than on versions. Interestingly, this cannot be possibly inferred by observing their grammars, which look similar at a first glance. Manual examination revealed that  $V_1$  and  $V_3$  agree on the use

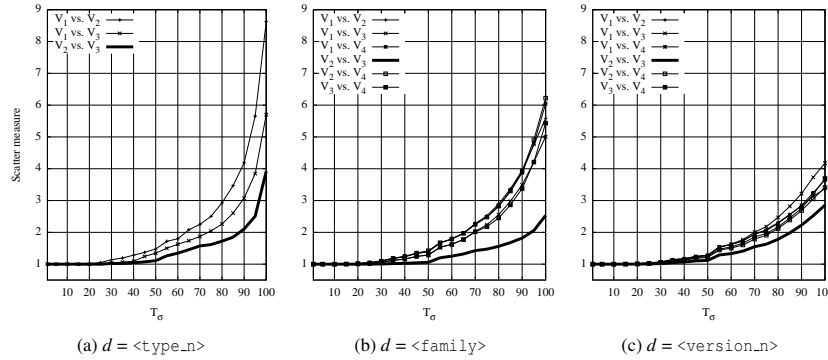


Figure 6: Scatter measure between each two vendors at different values of  $T_\sigma$ . At  $T_\sigma = 1.0\%$ , the comparison is optimistic as almost no coverage is required to find matching clusters between vendors; at  $T_\sigma = 100\%$  the comparison is realistic because, in order to match clusters between vendors, complete coverage is required. On average, almost every vendor have clusters that scatter onto 2–5 clusters of another vendor. Vendors  $V_2$  vs.  $V_3$  exhibit a steady scatter measure within 1–4, confirming their high degree of consistency according to Fig. 5.

of the keyword ``.gen`` to indicate the use of “generic” malware signatures. A negligible minority of samples were labeled with an additional progressive number (e.g., ``.gen44``) by  $V_3$ , which cannot be safely considered as proper version of the malware (but rather a signature identifier).

**4.3.3 Structural comparison** The connected components of graphs  $\mathcal{G}^{AB}$ , constructed by linking corresponding clusters between  $\mathcal{C}^A$  and  $\mathcal{C}^B$  (as described in §3.3.2) are good spots for finding consistencies, weak inconsistencies or strong inconsistencies among cross-vendor naming conventions. Recall that, as shown in Fig. 2, consistencies contain exactly two nodes (i.e., clusters), whereas weak and strong inconsistencies comprise several nodes. *Weak inconsistencies* are 1 :  $N$  relationships, where  $N$  indicates the granularity of one vendor with respect to the other, and by no means indicate a “badness” of an inconsistency. For example, if two vendors have a weak inconsistency with 4 nodes (1:3), this means that one vendor uses 3 different specific labels, while the other one uses only one label to indicate the same group of malware samples. Contrarily, *strong inconsistencies* are many-to-many relationships, and the number of nodes involved are a good indicator of significance of the inconsistency. This is because the more nodes are present in a connected component, the more complex the web of relationships between vendors’ labels is. For this reason, when we run our algorithm to extract connected components, we also calculated the size of those components leading to strong inconsistencies. It is important to underline that many small strong inconsistencies are better than only one big, strong inconsistency, because small inconsistencies can be easily visualized, analyzed, and manually reduced to weak inconsistencies (e.g., by removing one or two nodes). We repeated this experiment for  $T_W \in \{0\%, 10\%, 20\%, 40\%\}$ , i.e., by removing edges with weight below  $T_W$  from the weighted graphs. At  $T_W = 0$  the comparison is pessimistic and not quite realistic, because outliers in the dataset may create spurious links, not reflecting the overall characteristic of naming conventions, thus leading to the wrong conclusion that many strong inconsistencies exist. Also, high values of the threshold, e.g.,  $T_W > 50\%$ , may produce biased, and too much optimistic, conclusions, because relevant relations between naming conventions would be excluded from the analysis.

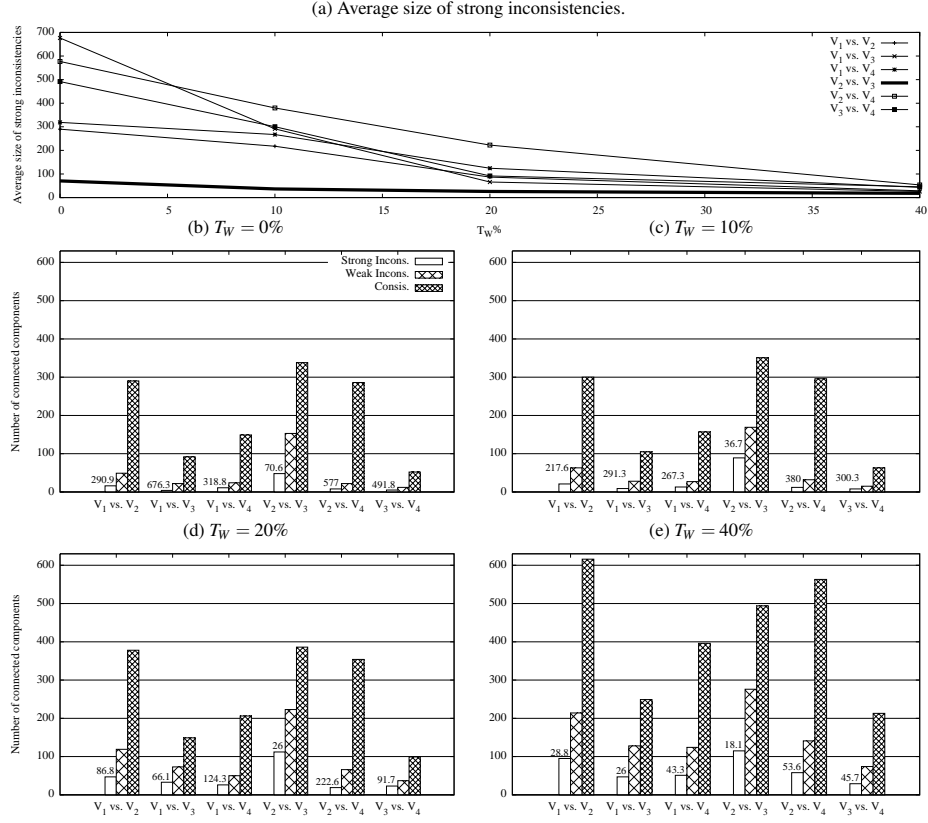


Figure 7: Number of structural consistencies compared to strong and weak inconsistencies for different values of the edge weight threshold,  $T_W$  (see §3.3.3). For strong inconsistencies, the average number of inconsistent clusters (i.e., those forming the graph’s connected component) is reported. Note that, several small inconsistencies are preferable (because easier to analyze and resolve) as opposed to one, large inconsistency.

Fig. 7a shows the average size of strong inconsistencies for different values of  $T_W$ . Interestingly,  $V_2$  vs.  $V_3$  is once again recognized as the most consistent couple of vendors. It indeed has the lowest average size of strong inconsistencies, ranging from 18.1 to 70.6. From Fig. 7(b-e) it can be noted that  $V_2$  vs.  $V_3$  have the highest number of consistencies (for  $T_W < 40\%$ ) and inconsistencies, indicated that their graph is well-fragmented in many small consistencies and many small inconsistencies.

This experiment shows that, although inconsistencies are generally more infrequent than consistencies, the number of strong inconsistencies is non-negligible. This result is magnified by the fact that the average size of strong inconsistencies is quite high. For instance, even at  $T_W = 40\%$  some vendors have strong inconsistencies comprising up to 53.6 nodes (average). It is interesting to compare this observation with Fig. 6 (scatter measures). Specifically, if one considers only the scatter measures, the average number of clusters that are scattered (across vendors) onto many different clusters is rather low. However, despite scatter is quite limited (e.g., less than 5 clusters for some vendors), it often yield strong inconsistencies. This is a consequence of the fact that scatter from  $A$  to  $B$  and *also* from  $B$  to  $A$ , thus yielding complex interconnections.



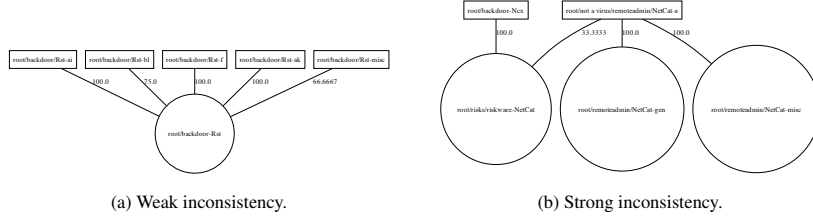


Figure 8: A real instance of a weak inconsistency (a) and strong inconsistency (b) between  $V_2$  (rectangles) and  $V_3$  (circles), which are the best-matching found. Interestingly, this randomly-selected weak inconsistency shows a case of name specialization, i.e.,  $V_2$  uses more fine-grained labels than  $V_3$ . Note that, we omitted the “not detected” clusters.

*Examples of extracted inconsistencies:* Fig. 8 shows two representative real cases of strong and weak inconsistencies, randomly extracted among the connected components of  $\mathcal{G}^{AB}$ , where  $A = V_2$  (rectangles) and  $B = V_3$  (circles), for  $T_W = 0\%$ . As mentioned in §3.3.2, weak inconsistencies indicate different granularities used by vendors to label malware samples. In the lucky case shown in Fig. 8a this is particularly evident by simply looking at the labels. However, labels do not typically reflect strong inconsistencies, which are less trivial. Notably, the example in Fig. 8b exhibits a strong inconsistency that would have been difficult to find by analyzing the labels, also because it involves different families, e.g., family *NetCat* belongs to two different types (i.e., *riskware* and *remoteadmin*) for the same vendor.

## 5 Limitations

The pattern extraction step of **Phase 1** may require manual intervention to decide the most appropriate class (e.g., `<type>`, `<family>`) for each encountered pattern. However, we implemented the extraction algorithm described in §3.2.1 and 3.2.3 once and adapted it for each vendor with minimal variations, mainly due to the heterogeneity of the `<type>` pattern class. Even without vendor support, we were able to cut down the number of manually-analyzed labels to a few tenths. To overcome this limitation entirely, community efforts or substantial support from antivirus vendors would be needed, but even as it is, the process is completely feasible.

Also, our technique provides a static snapshot of each vendor’s naming convention, at a given point in time. As reported in [4], malware naming conventions may change over time and, in addition, malicious code with new labels is unleashed with a relatively high frequency by the miscreants. To overcome this limitation, the structural models could be modified to incorporate a notion of “evolution” of a naming convention and quantitative measures should account be updated accordingly. Interestingly, this would allow to create and analyze a series of snapshots over time and, possibly, to support researchers at predicting future threats’ trends.

Last, a limitation of our experiments, and by no means of the proposed technique, is due to the fact that VirusTotal uses command-line antivirus engines, which may have different detection capabilities from their GUI-based equivalent, as observed in [4]. However, the importance of VirusTotal in our experiments is that it allowed us to query a quite extensive collection of malware samples, ranked by the number of times each sample has been scanned, which reflects the degree of “interest” around a virus.

## 6 Conclusions

Our proposed method is useful for finding inconsistencies as well as for comparing classifications (e.g., a ground truth vs. a classification produced by a novel approach being tested) by means of the number of inconsistencies contained.

Our experiments extended the previous results with interesting findings. First, we identified a number of cases of strong inconsistencies between classifications, demonstrating that the problem is structural, and not just in syntactic differences. A non-intuitive result is that, when a vendor's cluster is inconsistently mapped onto several clusters of another vendor, trying to map back those clusters to the first vendor spreads the inconsistencies even further. In other words, there is no guarantee that we will be able to identify a closed subset of malware on both sides that can be mapped consistently.

Our analysis shows that some vendors apply classifications that are completely incoherent (not just syntactically), but even between those who apply comparable classifications, inconsistencies are prevalent, and meaningfully mapping cannot be established. In simpler words, inconsistencies are not a case of the same family being labeled differently, but are structural and almost impossible to resolve. This also entails, in our point of view, that any usage of such classifications as a ground truth for clustering techniques or other automated analysis approaches should be carefully evaluated.

Besides addressing the limitation described in §5, future work include testing our method on more vendors, and in trying to derive the root-causes for the “closeness” of some vendors models. In addition, specific modeling of keywords denoting particular generations of malware samples, as well as packing techniques, need to be formalized.

## References

1. Andreas Marx, F.D.: The wildlist is dead, long live the wildlist! (2007), available online at [http://www.sunbelt-software.com/ihs/alex/vb\\_2007\\_wildlist\\_paper.pdf](http://www.sunbelt-software.com/ihs/alex/vb_2007_wildlist_paper.pdf)
2. Bailey, M., Oberheide, J., Andersen, J., Mao, Z.M., Jahanian, F., Nazario, J.: Automated classification and analysis of internet malware. In: Proc. of the 10th intl. conf. on Recent advances in intrusion detection. pp. 178–197. RAID'07, Springer-Verlag, Berlin, Heidelberg (2007)
3. Bontchev, V.: Current status of the caro malware naming scheme. Available online at <http://www.people.frisk-software.com/~bontchev/papers/naming.html> (2010)
4. Canto, J., Dacier, M., Kirda, E., Leita, C.: Large scale malware collection: Lessons learned (2008)
5. Carr, J.: Inside Cyber Warfare: Mapping the Cyber Underworld. O'Reilly Media, Inc., 1st edn. (2009)
6. Cova, M., Kruegel, C., Vigna, G.: Detection and analysis of drive-by-download attacks and malicious javascript code. In: WWW '10. pp. 281–290. ACM, New York, NY, USA (2010)
7. Gordon, S.: Virus and vulnerability classification schemes: Standards and integration. Tech. rep., Symantec Security Response (2003)
8. Harley, D.: The game of the name malware naming, shape shifters and sympathetic magic (2009)
9. Kelchner, T.: The (in)consistent naming of malware. *Comp. Fraud & Security* (2), 5–7 (2010)
10. Lancichinetti, A., Fortunato, S., Kertész, J.: Detecting the overlapping and hierarchical community structure in complex networks. *New Journal of Physics* 11(3), 033015 (2009)
11. MacKay, D.J.: Information Theory, Inference, and Learning Algorithms. Cambridge University Press (2003)
12. Mark K. Goldberg, Mykola Hayvanovych, M.M.I.: Measuring similarity between sets of overlapping clusters (Aug 2010)
13. Tarjan, R.: Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing* 1(2), 146–160 (1972)