

Finding Non-trivial Malware Naming Inconsistencies^{*}

Federico Maggi Andrea Bellini Guido Salvaneschi Stefano Zanero

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—more for researchers than for practitioners—for example, when comparing coverage of different antivirus engines, when integrating and systematizing known threats, or comparing the classifications given by different detectors. Clearly, solving naming inconsistencies is a very difficult task, as it requires that vendors agree on a unified naming convention. More importantly, solving inconsistencies is impossible without knowing exactly where they are. Therefore, in this paper we take a step back and concentrate on the problem of finding inconsistencies. 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., beyond syntactic differences) 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 that cannot be inferred at all by looking solely at the syntax.

1 Introduction

The current threat landscape is characterized by money-driven campaigns [1] mainly spread through drive-by download [2], more than by self-replicating code. 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 mutated versions of known malware, spurred a transformation in the mechanisms of action of antiviruses, which rely more and more on generic signatures and heuristics [3].

Because of historical reasons and vendor-specific policies, malware naming has never followed any conventions [3] (e.g., vendors and researchers used to name viruses based on characteristic they found interesting). Unfortunately, naming inconsistencies are a real problem when trying to correlate useful data across different antiviruses. Even simple problems such as comparing top-ten threat lists are in turn very difficult¹. Consequently, researchers have concentrated on solving inconsistencies and proposed both pragmatic approaches (e.g., VGreg, Wild List ²) and new naming conventions (e.g., CARO ³).

^{*} This is a pre-print version, which is an exact replica of the original article, except for the Acknowledgments section.

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

² http://www.sunbelt-software.com/ihs/alex/vb_2007_wildlist_paper.pdf

³ <http://www.people.frisk-software.com/~bontchev/papers/naming.html>

However, *finding* and *understanding* naming inconsistencies is a necessary (and missing) step before *solving* them. To address this, we extend the notion of “consistency” presented in [4], and propose a systematic approach for quantifying and finding inconsistencies. By observing these inconsistencies, and armed with the knowledge of a vendor’s detection methodology, an expert can investigate the inconsistencies.

We experimentally identify a number of strong inconsistencies, confirming that the problem is deep and structural. Also, we show that inconsistencies are not uniformly spread across different antiviruses (i.e., some vendors are more consistent, while others are wildly different). Last, we find large groups of inconsistently-labeled samples which cannot be made consistent in any sensible way.

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).
- We propose two quantitative measures that evaluate the overall degree of inconsistency between two vendors’ naming conventions (§3.3) and, more importantly, we define a simple algorithm that finds the inconsistent portions of graph model.
- 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 (§4), and briefly explain how these can be solved (§3.3).

2 Malware Naming Inconsistencies

Although variants of viruses and worms were relatively common, they tended to form just a small 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 across different vendors (e.g., with VGrep⁴). Even polymorphic viruses did not pose a serious challenge in this scheme. An effort to standardize names was CARO, which proposed the following naming convention: `<type>://<platform>/<family name>-<group name>.<length>.<sub variant><devolution><modifiers>`. However, this effort was unsuccessful. Even if it had been, a standard syntax would solve just a subset of the problem, without reconciling different family or group names between vendors.

The CME initiative⁵ tried to deal with the 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 result, many samples may descend from a mixture of ancestors, creating complex phylogenies that are 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. Consequently, correlating the outcomes of different antiviruses is even more complex [5]. For instance, in [4] signature-based antiviruses are compared with behavioral classifiers

⁴ <http://www.virusbtn.com/resources/vgrep/index.xml>

⁵ <http://cme.mitre.org/cme/>

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).

However, as of today, no complete and reliable method exists to find inconsistencies. Therefore, before consolidating malware names, we first need to quantify and spot them precisely.

3 Finding Naming Inconsistencies

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 (henceforth named “vendors” for simplicity). Our goal is to spot inconsistencies that go beyond well-known syntactic differences. Given a list of unique samples, our method produces a *qualitative* comparison, finds subsets of samples labeled inconsistently, and produces two *quantitative* indicators of the degree of inconsistency.



Phase 1 (modeling) For each vendor, we model malware names according to structural similarity between their *labels* (§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, when patterns such as “-backdoor”, “.backdoor.”, “-backdoor-dialer”, or “backdoor.dialer” are found, we assume that, according to the vendor under examination, the samples are all characterized by being “backdoors”, and thus **Phase 1** organize them in the same group. In other words, we model each vendor by means of the groupings induced by its naming convention. 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.

3.1 Types of Inconsistency

We define two different types of inconsistencies:

Weak inconsistency: One vendor divides the set of samples into more groups, whereas the other vendor groups them all together, thus 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 .

In §3.3 we further formalize these definitions by means of models constructed in **Phase 1**, and define a fast algorithm to identify them.

3.2 Phase 1: Naming Convention Modeling

We use a simple, top-down hierarchical approach (§3.2) to transform the initial set of malware samples into nested sub-sets, such that each (sub-)set contains only samples labeled with similar *string patterns*. Patterns are extracted offline for each vendor (§3.2).

Pattern Extraction Our technique is centered around four *pattern classes*, marked with angular brackets (i.e., `<class>`):

- `<type>` distinctive activity of the malicious code (e.g., “backdoor”, “worm”, or “dialer”, “packed”, “tool”).
- `<family>` name of a specific malware family (e.g., “Conficker”, “Mudrop”, “Fokin”, “Allaple”).
- `<platform>` operating system (e.g., “W32”, “WNT”) or language interpreter (e.g., “JS”, “PHP”).
- `<version>` malicious code version (e.g., “B” and “D” in labels “PHP : IRCBot-B” and “PHP : IRCBot-D”), or information to disambiguate “releases” or signatures (e.g., “gen”, “gen44”, “damaged”).

This small, generic set of pattern classes allows to analyze several vendors. New classes can be added and extend our approach to virtually any vendor. Our analysis on real samples revealed that each class can contain either one simple pattern or a hierarchy:

Simple pattern: occurrence of a string of a given class, e.g., `<type> = Trojan`. Usually, `<platform>` and `<family>` occur as simple patterns (e.g., `<platform> = Win32|PHP`).

Hierarchy: 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>`. We use the slash separator just to describe our results, though it by no means reflects any syntax.

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, as manual revision would be required anyway to ensure accurate results, we opt for a heuristic approach (detailed in §3.2), 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 `<typeX>` and `<typeY>`, $X < Y$ if `<typeX>` occurs more than `<typeY>` on a given set of malware sample. If they have the same frequency, the hierarchy is replaced by a simple pattern `<type>`, which contains the common substring between `<typeX>` and `<typeY>`.

Tree-based Models Given a set \mathcal{M} of labeled samples, we run the following procedure for each vendor on a given set of patterns. We consider one class of patterns at time.

We first split the initial set of labels according to `<type>`. For example, given `<type> = “Backdoor|Trojan”`, the samples labeled as `Backdoor.Pperl.Shellbot.cd`, `Backdoor.PHP.Shellbot.v` and `Backdoor.PHP.Shellbot.t` fall in the same sub-set, whereas `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

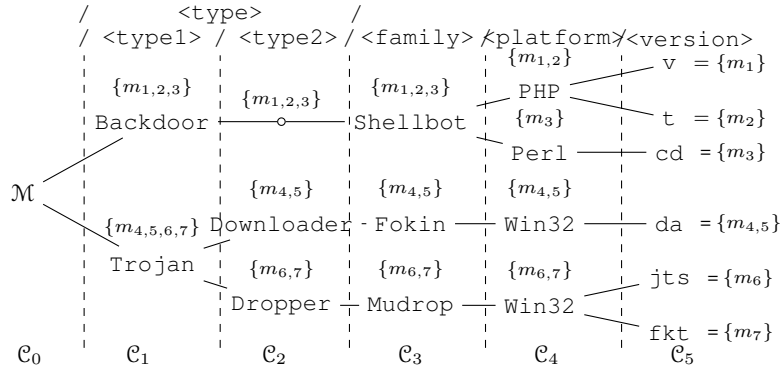


Figure 1: Example output of **Phase 1** on a set \mathcal{M} with seven samples. For instance, $\mathcal{C}_1 = \{\{m_1, m_2, m_3\}, \{m_4, m_5, m_6, m_7\}\}$ and $\mathcal{C}_{2-4} = \{\{m_3\}, \{m_6, m_7\}, \{m_4, m_5\}, \{m_1, m_2, m_3\}, \{m_1, m_2\}\}$. Note that, $\langle \text{type} \rangle$ comprises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

step is repeated for each sub-pattern. Continuing the above example, the trojan samples are separated in two different sub-sets.

When a (sub-)set can be split no further according to the same pattern class, we consider the $\langle \text{family} \rangle$. In our example, the only possible split is by means of “Fokin” and “Mudrop”, as “Shellbot” induces no splits. Then the first set is split in two sub-sets, 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-sets as well as “.fkt”, and “.jts” do.

We stop when the latest pattern class has been considered. In our example, the procedure ends after one split induced by the $\langle \text{version} \rangle$. At each split, we store the links between the sets thus constructing a tree, rooted in the initial set, as exemplified in Fig. 1.

Definition 1 (Naming Tree). Given a set \mathcal{M} of malware names, we define naming tree the output of **Phase 1**, which is $\mathcal{C}_d(\mathcal{M}) \subset \wp(\mathcal{M})$, 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 when implicit from the context).

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. We indicate portions of the naming tree with mnemonic expressions; for instance, “ $\langle \text{family} \rangle / \langle \text{platform} \rangle$ ” denotes the portion of naming tree corresponding to the $\langle \text{family} \rangle$, that are $\mathcal{C}_3 = \mathcal{C}_{\langle \text{family} \rangle / \langle \text{platform} \rangle} = \{\{m_1, m_2, m_3\}, \{m_4, m_5\}, \{m_6, m_7\}\}$. Actual substrings can be used as well: $\mathcal{C}_{\text{Backdoor} / \langle \text{platform} \rangle} = \{\{m_1, m_2, m_3\}\}$. A hierarchy of patterns always refer to the lowest depth. For instance, $\mathcal{C}_2 = \mathcal{C}_{\langle \text{type} \rangle / \langle \text{platform} \rangle} = \{\{m_1, m_2, m_3\}, \{m_4, m_5\}, \{m_6, m_7\}\}$.

Implementation Details

Pattern Extraction: The 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 iteratively breaks labels into substrings. At each iteration an operator reviews a set of candidate substrings and assign them to an 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 broken 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., $\langle \text{platform} \rangle_i = \text{Win32} | \text{PHP}$, $\langle \text{type} \rangle_i = \text{Trojan}$, $\langle \text{family} \rangle_i = \text{Allapple}$. 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 (\mathcal{L}_{i+1} may include postponed labels). The runs 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, as 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).

Singletons: Consider patterns $\langle \text{version} \rangle = v|t$ and a set $\{m_1, m_2\}$, where $m_1 = \text{Backdoor.PHP.Shellbot.v}$, $m_2 = \text{Backdoor.PHP.Shellbot.t}$. A split would produce two sub-sets $\{m_1\}, \{m_2\}$.

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 analyze consistency, we expect that, if two vendors are consistent, they would produce similar sets, also including “outliers”. For this reason, to take into account both the observations, sets of size below a certain threshold, T_o , are labeled with a special pattern, $\langle \text{misc} \rangle$ that encode such “uncertainty”.

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

Named Depth: For different vendors, a named depth (e.g., “ $\langle \text{family} \rangle$ ”), may correspond to different numerical depths. Therefore, while constructing the naming trees we construct the mapping between numerical and named depths.

3.3 Phase 2: Comparing Vendors

We compare two vendors A, B by means of their naming trees $\mathcal{C}^A, \mathcal{C}^B$ (Def. 1). We first calculate two indicators (§3.3) that quantify the degree of inconsistency between naming conventions between vendors; and then we spot inconsistencies (§3.3).

Naming trees are hierarchies of sets. However, we compare sets derived by “cutting” naming trees at a given depth d , omitted for simpler notation: $\mathcal{C}^A = \mathcal{C}_d^A$ and $\mathcal{C}^B = \mathcal{C}_d^B$.

Quantitative Comparison We define the *naming convention* distance, which expresses the overall difference between the naming conventions of A and B , and the *scatter measure*, which expresses the average number of sets of one vendor that are necessary to cover each set 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 sets.*

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

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

The denominator is such that $D(\cdot, \cdot) \in [0, 1]$, and $d(c, c') = 1 - J(c, c') \in [0, 1]$, where $J(c, c')$ is the Jaccard index. A similar distance was used to measure the similarity between sets of overlapping trees of sets [6].

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

In other words, Γ 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). As \mathcal{C}^A and \mathcal{C}^B are *partitioned*, $|\Gamma(c, \mathcal{C}')|$ is the number of sets of \mathcal{C}' that build c .

Structural Comparison We recognize the inconsistencies defined in §3.1. To this end, trees at a given depth are first represented as undirected graphs with cross-vendor edges, and then searched for inconsistent sub-graphs. This comparison applies only when \mathcal{C}^A and \mathcal{C}^B are partitioned into flat sets. Therefore, only for this analysis, we assume that sets are drawn from trees at leaf depth (i.e., `<version.n>`), representative of the whole label.

Definition 5 (Linked Naming Trees). *Given \mathcal{C}^A and \mathcal{C}^B the linked naming tree is an undirected graph $\mathcal{G}^{AB} := \langle \mathcal{V}^{AB}, \mathcal{E}^{AB} \rangle$, where $\mathcal{V} = \mathcal{C}^A \cup \mathcal{C}^B$ and $\mathcal{E}^{AB} = \{(c, c') \mid c \in \mathcal{C}^A, c' \in \mathcal{C}^B \wedge c \cap c' \neq \emptyset\}$.*

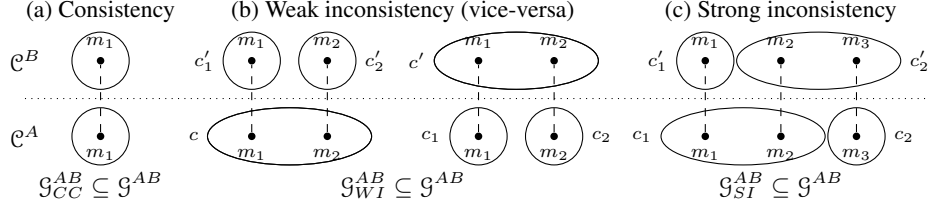


Figure 2: Instances of consistencies \mathcal{G}_{CC}^{AB} , weak inconsistencies \mathcal{G}_{WI}^{AB} and strong inconsistencies \mathcal{G}_{SI}^{AB} , i.e., connected components of the linked naming tree \mathcal{G}^{AB} of vendors A vs. B . Each vertical line represents a malware sample.

In other words, \mathcal{G}^{AB} encodes the links between sets of labeled samples. Given a set c of samples labeled by A , and a set c' of samples labeled by B , we set an edge from c to c' only if c' has at least one sample in common with c . In §3.3 we extend this concept with edges weighted proportionally to the number of samples shared between c and c' . Therefore, we reduce the problem of recognizing inconsistencies to finding connected components of \mathcal{G}^{AB} , for which efficient algorithms (e.g., [7]) exist. The connected components are then analyzed automatically to distinguish among:

Consistency (CC) (Fig. 2a) The connected component has two sets with the same samples (samples of A may have different labels than samples of B).

Weak Inconsistency (WI) (Fig. 2b) The connected component contains only one set $c \in \mathcal{V}^A = \mathcal{C}^A$, and all sets $c' \in \mathcal{V}^B = \mathcal{C}^B$ 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 in each label.

Strong Inconsistency (SI) The connected component contains more than one set for each vendor (e.g., for sets c'_1, c_1, c'_2, c_2 in Fig. 2c). As sets are partitions of the entire set of malware samples, there must be at least four sets $c_1, c_2 \in \mathcal{V}^A = \mathcal{C}^A$, $c'_1, c'_2 \in \mathcal{V}^B = \mathcal{C}^B$ such that the following condition holds: $c_1 \cap c'_1 \neq \emptyset \wedge c_2 \cap c'_2 \neq \emptyset \wedge c'_2 \cap c_1 \neq \emptyset$. In other words, the sets share some samples without being all subsets of each other. The inconsistency, which includes all sets of the connected component, is caused by inherently different naming conventions. Once found, these inconsistencies can be solved by fusing, say, c_1 with c_2 .

Implementation Details

Scatter Set Coverage: Our implementation incorporates a measure of *coverage*, σ , in scatter sets $\Gamma(c, \mathcal{C}')$ (Def. 4), defined as $\sigma(\Gamma) := |c \cap \bigcup \mathcal{C}'|/|c|\%$, where the union is calculated for any $c' \in \Gamma(c, \mathcal{C}')$. The coverage quantifies the percentage of samples in c (e.g., a set of vendor A) shared with the union of scatter sets derived from \mathcal{C}' (e.g., a set tree of vendor B). Scatter sets can be selected by their σ , 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 $\hat{\Gamma}_{T_\sigma} : \nexists \Gamma(c, \mathcal{C}') \text{ for } \sigma(\Gamma) \geq T_\sigma \wedge |\Gamma| < |\hat{\Gamma}|$: The smallest scatter set that covers c of at least T_σ .

Weighted Linked Naming Trees: The edges of the linked naming trees (Def. 5) are weighted with the following weighting function:

$$W(c, c') := \max \left\{ \frac{|c \cap c'|}{|c|} \%, \frac{|c \cap c'|}{|c'|} \% \right\}, \forall (c, c') \in \mathcal{E}^{AB} \quad (3)$$

Each edge encodes the degree of “overlapping” between two sets 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

Microsoft V_1 `<type>:<plat>/<family>[.gen[!<ver1>]|<ver2>]` 4,654 labels
 Antiy V_2 `<type>.<plat>/<family>[.<ver1>.<ver2>]` 23,603
 Kaspersky V_3 `<type>/<plat>.<family>[.gen]` 2,122
 Avast V_4 `<plat>/<family>[-gen|-<ver>]` 4,350

These naming conventions cover the vast majority of the samples. These vendors are good candidates because the “richness” of their naming convention allows a granular analysis, which spans from `<type>` to `<version>`. Adding more vendors is computationally feasible, although 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 our goal in this paper is to evaluate our method and show that it finds structural inconsistencies, as opposed to simply quantifying them, using four vendors, totaling six comparisons, seems sufficient and actually clearer.

Vendor V_4 includes no `<type>`. We manually analyzed this case and discovered that the `<family>`, which is instead present in the syntax, is seldom used also to hold information about the threat type (e.g., “Malware”, “Dropper”, “Trojan” in Fig. 4). As this happens quite seldom, 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 handle this exception by assigning `<family> = “generic”`.

Dataset: Our dataset \mathcal{M} , generated on Sep 13, 2010, comprises 98,798 distinct malware samples identified by their hashes. We derived the labels $\mathcal{L}^{V_1}, \mathcal{L}^{V_2}, \mathcal{L}^{V_3}, \mathcal{L}^{V_4}$ via VirusTotal, an online service which allows to scan samples with multiple vendors simultaneously. We selected a set of 98,798 samples recognized by the majority of the four main vendors. 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, about 0.10223% of the dataset) fall outside the above convention. From a purely syntactic point of view, these cases are similar to the presence of keywords

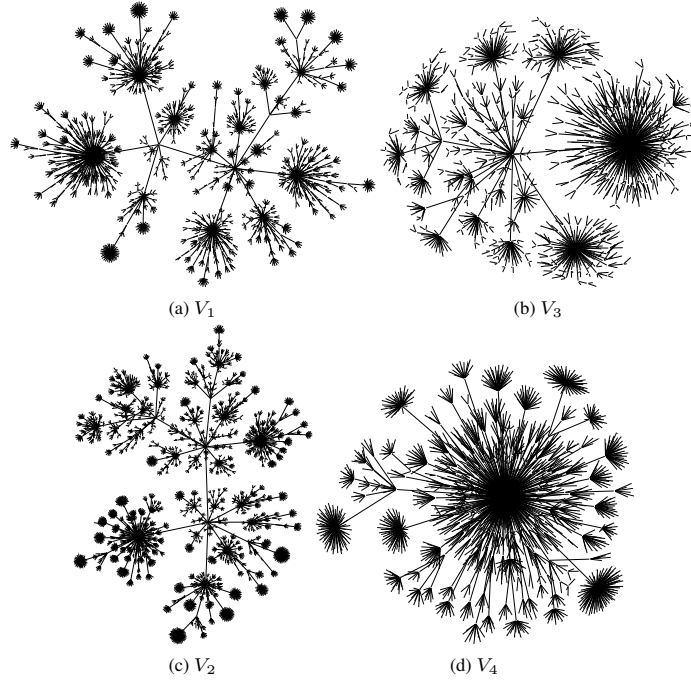


Figure 3: Visual comparison of naming trees of each vendor.

that often mark special samples (e.g., “packed”). We handled this handful of outliers manually.

Phase 1 was run on the dataset to create a model, i.e., naming 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 naming trees.

4.1 Naming Tree Visual Comparison

Different naming conventions induce naming trees that are structurally dissimilar, as evident even at a high level (Fig. 3): V_4 splits samples very early in many sub-sets based on their `<family>`, whereas other vendors use finer conventions and form sparser trees.

For ease of visualization, in Fig. 4 we extracted a slice of 50 samples from one set (i.e., classified as `packed-Klone` by V_3 , used as a comparison baseline). Vendor V_1 spreads the same samples onto 8 sets, 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 the labels’ granularity varies across vendors. For instance, V_1 , which adopts a granularity of 1 to 3, splits `worms` (depth 1) in `Pushbots` and `Miscellaneous` (depth 2)⁶. Instances of the same behavior occur in V_2 , with

⁶ In this example, singletons are visualized as such, but actually contain more than one sample; this is because we sliced a set of 50 samples from a full naming tree.



Figure 4: A flat set extracted from V_3 's naming tree cut at `<family>` depth. Other vendors group the same set of samples differently. Only one baseline set is shown, although the same behavior can be observed also with other baselines, i.e., V_1 , V_2 , V_4 . depths of 1 to 4, and V_4 , with depth is 1. We now analyze these “visual” discrepancies thoroughly.

4.2 Singletons and “not detected” Samples

Certain values of T_W may yield isolated singletons (i.e., singletons from one vendor with no corresponding sets in the counterpart). Depending on the structure of the linked naming trees, by varying T_W , these nodes may either link to another single node (i.e., consistency), or to several nodes (i.e., inconsistency). Optimistically, we could count them as consistencies because, for a certain, low value of T_W , at least one linked set exists. On the other hand, we could consider such nodes as potential inconsistencies. Due to this inherent ambiguity, we ignore singleton nodes to avoid biased results.

During pattern extraction, we treat samples that are not detected as malicious by one vendor as labels containing only a `<type_n>` string. These are not exactly naming inconsistencies, as they depend on detection accuracy more than on naming structures.

Indeed, they can originate from false positives or false negatives. These nodes link to several other sets, and thus spur a minority of very large inconsistencies—possibly up to the whole graph. This may bias the quantitative comparison. Hence, we removed such sets from the following analysis. More precisely, the scatter measure discussed in §4.3 ignores the scatter sets originating from these sets. Similarly, the linked naming trees (Def. 5) used for structural comparison, discussed in §4.3, was pruned by removing “not detected” sets (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

Naming Convention Distance Fig. 5a summarizes the distance for each unique couple of vendors A vs. B , quantifying the overall inconsistency between the vendors’ naming conventions. 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 naming tree would exhibit progressively lower consistency. Also, vendors V_2 and V_3 are remarkably more consistent than the other couples, especially at `<family>` depth. These two vendors exhibit small structural inconsistencies as also noted in §4.3.

Scatter Measure The scatter measure how elements of one set of vendor A are distributed (i.e., scattered) onto (multiple) sets of vendor B (and viceversa). We calculate the scatter measure at different values of coverage, $\sigma(I)$, of the scatter set (i.e., the set of sets in \mathcal{C}^B that corresponds to the set $c \in \mathcal{C}^A$ under examination, and vice versa). We do this from A to B and vice versa, and vary a threshold T_σ . Therefore, we calculate $S(\mathcal{C}^A, \mathcal{C}^B)$ for $T_\sigma \in \{1\%, 5\%, 10\%, \dots, 95\%, 100\%\}$. Low values of T_σ lead to lower, optimistic, values of $S(\cdot, \cdot)$, reflecting the existence of small scatter sets, which are selected even if they cover only a slight portion of the set under examination. Contrarily, higher values of T_σ unveil the *real* scatter sets, that are those with substantial overlapping.

Fig. 5(b–d) summarizes the results of this experiment for each couple of vendors at different “cuts” of the naming trees: $d \in \{\text{<type_n>}, \text{<family>}, \text{<version_n>}\}$. 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 their overall consistency, especially at `<family>` level—as we concluded from Fig. 5a.

Another interesting comparison is V_1 vs. V_3 , which, according to Fig. 5a, show remarkable distance and thus can be considered different from one another. First, Fig. 5(b–d) confirms this conclusion. In addition, these vendors tend to have divergent scatter measures (for increasing values of T_σ), especially at `<type_n>` depth (Fig. 5b), thus revealing that they disagree more on threat types than on versions. Interestingly, this cannot be inferred by observing their grammars, which look similar at a first glance. Manual examination reveals that V_1 and V_3 agree on the use of the keyword ‘`.gen`’ to indicate the use of “generic” malware signatures. A negligible minority of samples are labeled with an additional progressive number (e.g., ‘`.gen44`’) by V_3 , which cannot be safely considered as proper version of the malware.

Structural Comparison The connected components of the linked naming trees, \mathcal{G}^{AB} , constructed by connecting corresponding sets between \mathcal{C}^A and \mathcal{C}^B (as described in §3.3) are good spots for finding consistencies, weak inconsistencies or strong inconsistencies.

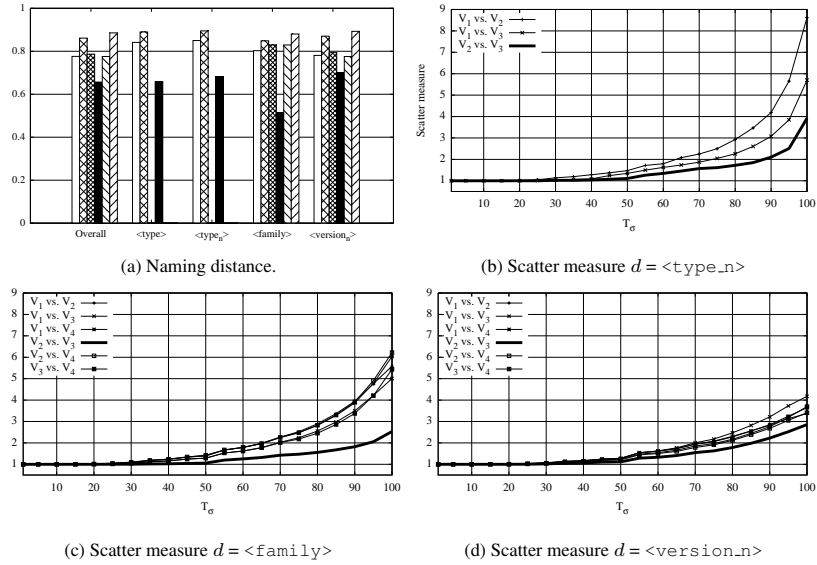


Figure 5: Naming convention distance (a) at different depths of the naming trees, and scatter measure (b–d) between each two vendors at different values of T_σ . Relatively high distance between vendors is observed. Notably, the depth (e.g., $\langle \text{type} \rangle$, $\langle \text{family} \rangle$) negligibly influences the distances, except for V_2 vs V_3 , which exhibit slightly more similarity in terms of $\langle \text{version} \rangle$. At $T_\sigma = 1.0\%$, the scatter measure is optimistic as almost no coverage is required to find matching sets between vendors; at $T_\sigma = 100\%$ the comparison is realistic because, in order to match sets between vendors, complete coverage is required. On average, almost every vendor have sets that scatter onto 2–5 sets 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 the naming distance (a).

As shown in Fig. 2, consistencies contain exactly two nodes (i.e., sets), whereas weak and strong inconsistencies comprise several nodes. Weak inconsistencies are 1-to- 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, a 1-to-3 weak inconsistency, simply means that one vendor uses 3 different labels, whereas the other vendor groups same malware samples in one set. Contrarily, strong inconsistencies are M -to- N relationships, and M or N are good indicators of the significance of the inconsistency: The more nodes are present in a connected component, the more complex the web of relationships between labels is. In general, many small, strong inconsistencies are better than one big, strong inconsistency: Small inconsistencies can be easily visualized, analyzed, and reduced to weak inconsistencies (e.g., by removing one or two nodes, or by fusing sets). We kept track of the size of the components that yield strong inconsistencies at different values of $T_W \in \{0\%, 10\%, 20\%, 40\%\}$, that is, we removed edges with weight below T_W from the weighted graphs. At $T_W = 0$ the comparison is unrealistic, as outliers may create spurious links, not reflecting the overall characteristic of naming conventions, thus leading to the wrong conclusion that many

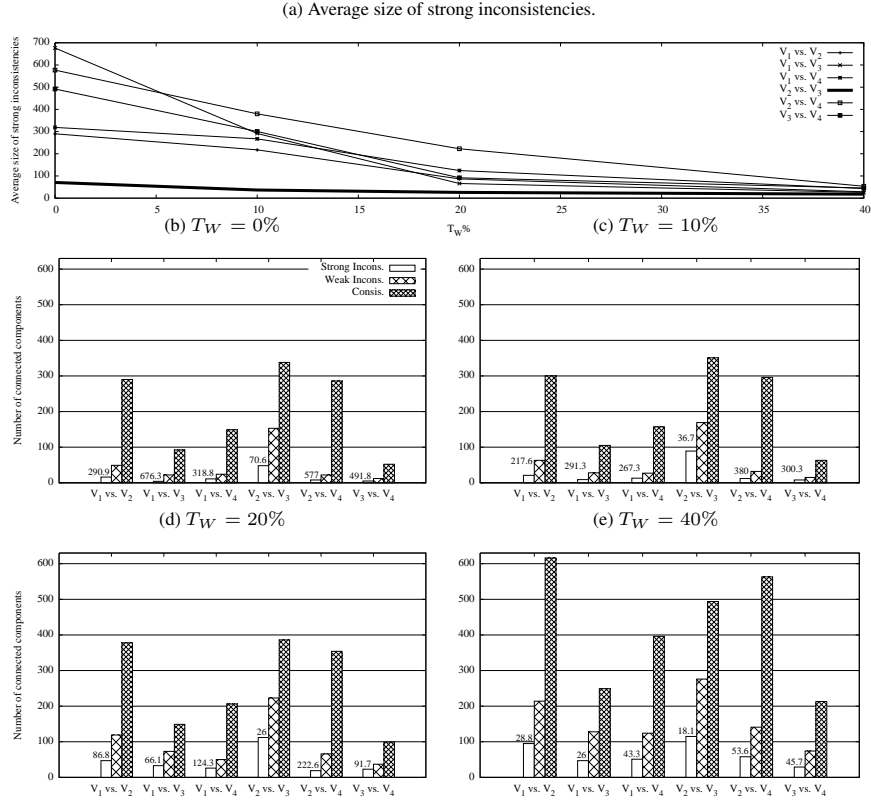


Figure 6: Number of structural consistencies compared to strong and weak inconsistencies for different values of the edge weight threshold, T_W (see §3.3). For strong inconsistencies, the average number of inconsistent sets (i.e., those forming the graph’s connected component) is reported. Note that, several small inconsistencies are preferable (because easier to analyze and resolve) as opposed to one, large inconsistency.

strong inconsistencies exist. Also, high values (e.g., $T_W > 50\%$) may produce biased (optimistic) conclusions, as relevant relations between naming conventions are excluded.

Fig. 6a shows the average size of strong inconsistencies: V_2 vs. V_3 are once again the most consistent vendors, with the lowest average size of strong inconsistencies (i.e., from 18.1 to 70.6). In Fig. 6(b–e), V_2 vs. V_3 show the highest number of consistencies (for $T_W < 40\%$) and inconsistencies, thus their graph is well-fragmented in many small consistencies and many small inconsistencies.

Although inconsistencies are generally more infrequent than consistencies, the number of strong inconsistencies is significant. This is exacerbated by the average size of strong inconsistencies, which is quite high. For instance, even at $T_W = 40\%$ some vendors have strong inconsistencies comprising up to 53.6 nodes on average. Comparing this observation with Fig. 5(b–d) (scatter measures), we note that the average number of sets that are scattered (across vendors) onto multiple sets is rather low. However, despite scatter is quite limited (e.g., less than 5 sets for some vendors), it often yields strong inconsistencies, because it occurs both from A to B and vice versa.

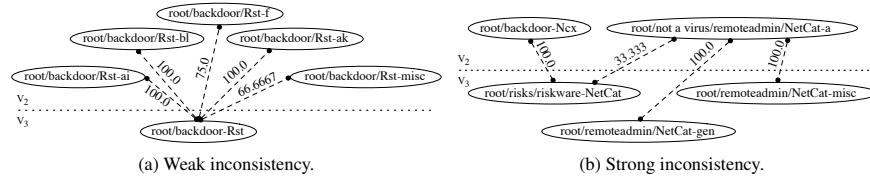


Figure 7: A real instance of a weak inconsistency (a) and strong inconsistency (b) between V_2 and V_3 , which are the best-matching found. This randomly-selected weak inconsistency shows a case of name specialization, where V_2 uses finer labels than V_3 .

Examples of Found Inconsistencies: Fig. 7 shows two representative, real cases of strong and weak inconsistencies between $A = V_2$ and $B = V_3$, for $T_W = 0\%$. As mentioned in §3.3, weak inconsistencies indicate different granularities used by the vendors’ labels that, in the lucky case of Fig. 7a, are easy to recognize. However, strong inconsistencies are less trivial to spot by comparing labels, as shown in Fig. 7b: This strong inconsistency is difficult to find by analyzing the labels, also because it involves multiple families (e.g., NetCat belongs to two different types: *riskware* and *remoteadmin* for the same vendor).

5 Conclusions

Our 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. A non-intuitive result is that, when a vendor’s set is inconsistently mapped onto several sets of another vendor, trying to map back those sets to the first vendor spreads the inconsistencies even further. In other words, there is no guarantee that a closed subset of malware on both sides that can be mapped consistently exists. 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.

Future work may address the fact that pattern extraction sometimes requires manual intervention to assign string patterns to appropriate. However, without vendor support, we had to manually analyze only a few tenths of labels. This limitation could be mitigated with community-driven efforts. Also, as malware naming conventions may change over time, we should incorporate a notion of “evolution” of a naming convention.

6 Acknowledgments

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreements nr. 257007 (SysSec) and 216026 (WOMBAT). The opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the European Commission. We gratefully acknowledge VirusTotal for the access to their malware database.

References

1. Carr, J.: Inside Cyber Warfare: Mapping the Cyber Underworld. O’Reilly Media, Inc. (2009)
2. Cova, M., Kruegel, C., Vigna, G.: Detection and analysis of drive-by-download attacks and malicious javascript code. In: WWW, New York, NY, USA, ACM (2010) 281–290
3. Kelchner, T.: The (in)consistent naming of malware. *Comp. Fraud & Security* (2) (2010) 5–7

4. Bailey, M., Oberheide, J., Andersen, J., Mao, Z.M., Jahanian, F., Nazario, J.: Automated classification and analysis of internet malware. In: RAID. (2007)
5. Harley, D.: The game of the name malware naming, shape shifters and sympathetic magic. In: CEET 3rd Intl. Conf. on Cybercrime Forensics Education & Training, San Diego, CA (2009)
6. Mark K.Goldberg, Mykola Hayvanovych, M.M.I.: Measuring similarity between sets of overlapping clusters. In: SocialCom, Minneapolis, MN (Aug 2010)
7. Tarjan, R.: Depth-First Search and Linear Graph Algorithms. SIAM J. on Comp. **1**(2) (1972)