The Polymorphism Maze: Understanding Diversities and Similarities in Malware Families

Antonino Vitale[1]✉[0009-0002-8452-4532], Simone Aonzo[1][0000-0001-9547-3502], Savino Dambra[2][0000-0002-0988-9366], Nanda Rani[3][0000-0003-1255-5284], Lorenzo Ippolito[1][5][0009-0008-7978-6777], Platon Kotzias[4][0000-0003-3375-6069], Juan Caballero[5][0000-0003-2962-1348], and Davide Balzarotti[1][0000-0001-5957-6213]

[1] EURECOM, France
`{Antonino.Vitale, Simone.Aonzo, Davide.Balzarotti}@eurecom.fr`
[2] GenDigital, France
`savino.dambra@gendigital.com`
[3] Indian Institute of Technology Kanpur, India
`nandarani@cse.iitk.ac.in`
[4] BforeAI, USA
`platon@bfore.ai`
[5] IMDEA Software Institute, Spain
`juan.caballero@imdea.org`

Abstract. In this work, we explore the complexities introduced by polymorphism in malware families, a tactic used by malware authors to alter the appearance of their code and evade detection mechanisms, resulting in a growing volume of unique malware samples. We examine 66,160 malicious Portable Executable (PE) files grouped into 743 families from three popular malware datasets. Our research addresses three key questions: measuring structural component-level differences between PE files, identifying prevalent polymorphic techniques affecting multiple components, and pinpointing component-level causes of polymorphism.
We introduce a methodology for component-level structural comparison of PE files and apply it to investigate the diversity and similarity of samples within a family, considering factors such as packing and truncation. Our study reveals that polymorphism in malware is driven by multiple overlapping factors, extending beyond just the use of packing tools. These findings highlight the complex nature of malware families and inform future research, improving our understanding of malware variations and their implications.

Keywords: Malware Polymorphism · Static Analysis · Malware Similarity

1 Introduction

What is a *malware family*? Despite decades of research and thousands of papers on malware, the scientific community lacks a precise definition. One possible definition is that a malware family consists of different malicious samples (i.e., different by file hash) derived from the same code base, analogous to how a *program* is defined in the benign case. Samples within a family are expected to share common characteristics, behaviors, and attribution to the same authors [1].

However, malware authors often adopt a wealth of techniques to introduce *polymorphism* (e.g., by re-packing samples) so that the same malware family version produces a large number of *variants* that are derived from the same source code but differ in their representation, i.e., have a different file hash. Polymorphism can also be introduced in other ways. For example, file infectors inject their code into other benign executables. Because of this, polymorphism is often cited as the main reason behind the large number of new malware samples routinely collected and analyzed by the security industry [16].

In summary, many reasons can be behind the differences (namely, polymorphism) between samples within the same family. What is more important is that these differences are not just a curiosity, but they also impose severe consequences for the analysis, detection, and classification of malware. For instance, we expect AntiVirus (AV) signatures to capture not a single sample but an entire family (or a part of it). Similarly, we expect ML models trained to recognize a given family to succeed when tested on new samples that belong to the same family. But this ability to generalize largely depends on **why** samples differ in the first place. Small differences in the PE header do not have the same impact as re-packing the code with a different protector.

This paper aims to provide the first comprehensive exploration of the reasons behind the polymorphism in the samples belonging to the same malware family. For this, we leverage three malware family datasets (the recently-proposed dataset by Dambra et al. [15], the MOTIF dataset [3], and the Malicia dataset [33]), building a superset composed of 66,160 samples split into 743 families. Our analysis involves a static examination of malware samples to understand the syntactic variations within samples belonging to the same malware family. Static analysis is preferred for examining such syntactic characteristics, whereas dynamic analysis is necessary for assessing behaviors. However, in our dataset, the behavioral similarities among the samples are already captured by the family labels assigned to the samples. More specifically, our work is organized to answer the following three Research Questions:

RQ1: *How can we measure the structural differences among multiple samples from the same family?* At first, in Section 3, we break down the PE file format in a number of disjoint *components*, that fully cover the whole PE file format structure and content. Then, given two executables, we design a structural comparison approach to precisely locate their differences and similarities at the component level. We implement our approach in an open-source tool we named *PEdiff* [4].

RQ2: *What are the polymorphic techniques that affect multiple components, and what is their prevalence?* In Section 4, we examine two main reasons for cross-component differences: file truncation and packing. *Truncation* occurs when the expected size of a sample is larger than the real size of the file on disk. Truncation occurs due to errors during sample collection (e.g. samples extracted from network traffic where packets were missing). *Packing* is a technique that compresses or encrypts code on disk and then recovers it at runtime. We measure packing in two ways: by using state-of-the-art signature-based tools to reliably detect known off-the-shelf packers and by implementing a machine learning (ML) classifier proposed by Aghakhani et al. [6] to also identify custom packers.

RQ3: *What are the many reasons of polymorphism at the component granularity?* In Section 5, we examine polymorphism in one or multiple components. Our results show that two-thirds of the families have no common components among their samples, meaning that all the PE components are at least slightly different. On the other hand, for 12.8% of the families, we were able to pinpoint the single reason behind the polymorphic variants.

In summary, we first developed a novel methodology for the structural comparison of PE files. Then, we highlighted the importance of two common elements (packing, truncation) which are crucial for the construction of malware datasets. We advocate for the community to conscientiously consider the elements they wish to exclude or include in their studies, given the potential bias these decisions may introduce. Lastly, we conducted a comprehensive measurement of polymorphism across 743 malware families. This analysis provides valuable insights for future research, enabling a targeted focus on the most prevalent trends and the timely development of appropriate solutions.

Finally, the scientific significance of this work is particularly relevant in the context of the design and evaluation of robust ML classifiers: we believe that their (in)ability to generalize to different samples needs to be always corroborated by an analysis of the variability of samples within the families in the dataset.

2 Dataset

We use three datasets of malware samples, each consisting of Windows PE executables labeled with the family to which the sample belongs.

Dambra et al. [15]. We use their balanced dataset which contains 67,000 hashes of 32-bit PE files that appeared in the VirusTotal (VT) feed between August 2021 and March 2022. The samples are equally divided among 670 malware families, i.e., 100 samples per family. The family labels were obtained by processing the VT reports using AVClass [42]. We download the hashes and family labels from their repository [2] and then download the samples and their reports from VT.

MOTIF [25]. This dataset contains 3,095 PE malware samples from 454 families. Samples and family labels come from threat reports published by 14 major cybersecurity organizations between January 2016 and December 2020. We obtained the list of sample hashes and family names from the MOTIF repository [3] and then downloaded the samples from VT. MOTIF is largely imbalanced. Of the 454 families, 131 (29%) have only one sample, while only 91 (20%) have at least 10 samples.

Type	Components
Metadata	DOS Header, DOS Stub, Rich Header †, COFF Header, Optional Header, Data Directories †, Section Table
Sections	Entry Point Section, Resource Section †, Other Sections †
Extra	Certificate Table †, Overlay †

Table 1: Components of a PE executable and their type. The dagger † indicates an optional component.

Malicia [33]. The Malicia dataset contains 9,908 Windows PE malware samples collected from drive-by downloads between March 2012 and February 2013 [33]. Labels for the samples were generated by clustering the samples using network features and screenshots obtained during the sample's execution, and the embedded icon. We obtained the samples and family labels from the dataset authors. Malicia is also largely imbalanced with 23 (43%) families having only one sample, while only 13 (25%) have at least 10 samples.

Final dataset. We start with the Dambra et al. dataset and add families from MOTIF and Malicia with at least 10 samples, which we consider the minimum to analyze differences within a family. We exclude families already present in the balanced dataset. For families with over 100 samples we randomly selected 100 samples. This procedure outputs 68,683 samples split into 746 families.

As we elaborate in Section 4.1, we identified truncated samples in the datasets. We removed the truncated samples and also families with less than 10 non-truncated samples. In the end, the final dataset used in this study comprises of **66,160** samples distributed in **743** families.

3 Structural Comparison

While samples in a family differ in file hash, they may exhibit similarities, while their differences may be concentrated on specific parts. To examine similarities and differences within a malware family, we have designed a methodology for structural comparison of executables. It first divides each executable into 12 disjoint *components*, described in Section 3.1, that fully represent the PE executable format [30]. Next, it performs pairwise comparisons of all executables in a family at the component level, categorizing components as unchanged, similar, or different, as detailed in Section 3.2 We have implemented our methodology into *PEdiff* [4], an open-source tool comprising 1K lines of Python code.

3.1 PE Components

We split each executable into 12 disjoint (i.e., non-overlapping) components that capture its structure, depicted in Table 1. We grouped them into three parts:

the *Metadata* contains the first seven components, which do not carry the actual content of the executable but define its structure and properties, the *Sections* which contain the code and data of the executable, and the *Extra*, which consists of components that are appended at the end of the file. Of the 12 components, six are optional and may not exist.

Within the Metadata, the *DOS Header* and *DOS Stub* correspond to the legacy MS-DOS information that is still present for compatibility. The *COFF header* and *Optional Header* capture the homonymous PE headers. The *Rich Header* is an undocumented component containing information about the tool versions used to build the different object files in the executable [50]. The *Data Directories* is an array of 16 entries, where each entry contains the start offset and size of a data directory, including the export, import, resource, and certificate tables. The *Section Table* is an array that defines the name, start offset and the size of the sections that form the main body of the executable.

We identify three Sections components: The *Entry Point Section* is the section that contains the *AddressOfEntryPoint* field of the Optional Header. The *Resources Section* is a special section that contains a tree structure holding data items such as strings, images, and icons. Finally, the *Other Sections* component captures all other sections in the executable that do not contain the entry point or the resources. This is the only component that does not necessarily correspond to a contiguous sequence of bytes, since the order of the sections is defined in the Section Table and the entry point and resources sections may not be the first or last sections.

Executables may contain two optional Extra components. For signed executables, the *Certificate Table* contains a digital signature and a list of X.509 certificates for validating the file's integrity and the identity of the publisher. The *Overlay* component captures data appended at the end of an executable. This data is not described in the PE header, thus it is ignored by the loader. However, it is accessible by reading it directly from the file on disk. The presence of an overlay can be identified because the file's expected size (i.e., the sum of the start offset and size of the last section) is smaller than the real size of the file on disk. Some tools consider the certificate table to be an overlay. However, we consider it a separate component because its start offset and size are defined in the Data Directories and thus its existence is known to the loader. For signed samples, we consider that an overlay exists if and only if there is additional data after the end of the certificate table.

3.2 Family Component Analysis

Given the samples in a family, our goal is to identify which components are similar and different in the family. For this, we compare the contents of a component across all pairs of samples in the family. For each component in each pair of samples, we apply a pairwise similarity function to determine whether the contents of the component across the two samples are similar or not, and accumulate results across all pairs of samples.

Pairwise similarity. We experiment with three Boolean similarity functions that given the content of a component in two samples determine whether the component is similar. The first function computes the SHA256 hash of the sequence of raw bytes of the component[6] and checks if both hashes are the same. This is the strictest similarity function requiring both samples to have identical content in the component. The second function computes instead the TLSH [34] fuzzy hash over the components' raw bytes. Fuzzy hashes output similar digests when the inputs are similar. Among all the fuzzy hashes available in the wild, we chose TLSH because it is the one that can produce a hash for the smallest stream of bytes, given that the minimum size is 50 bytes. Thus, it can handle most of the smallest components that are usually headers. Other fuzzy hashes could require very large minimum sizes (e.g. SSDEEP requires at least 4KB to compute the hash). TLSH returns a distance in the $x \in [0, \infty)$ range, which we normalize ($y = max\{\frac{300-x}{3}, 0\}$) to a similarity in the $y \in [0, 100]$ range as suggested by other works [36, 47]. The component values are considered similar if the TLSH similarity was ≥ 90, as proposed by Oliver et al. [34]. This function is more lax because it considers the component values to be similar even if they are not identical, as long as the raw byte differences are small. Pagani et al. [36] showed that TLSH can remain robust when small modifications are introduced in the code; however, they also observed that compiling the exact same source with seemingly minor tweaks (such as slightly different compiler flags) can result in anything from negligible differences to extensive ripple effects in the final executable. Therefore, we compute the code similarity using the popular BinDiff [18] tool, which disassembles both executables (using IDA Pro 8.1 in our setup) and uses graph isomorphism and heuristics to match their functions. It returns a similarity value in $[0, 1]$. The advantage of BinDiff is that it disassembles the code and thus can ignore differences in the data between code blocks and handle some code reordering. But, it only measures code similarity, so we only apply it to the Entry Point Section component. We determine that the Entry Point Section of two samples is similar if their BinDiff similarity is > 0.85, as suggested by Egele et al. [17].

Family components. To determine if a component is similar, different, or missing across a family we use Algorithm 1. It takes as input the $10 \leq n \leq 100$ samples that belong to a family, a similarity function, and a threshold t. For each component c, it initializes to zero two counters: C_d^c and C_p^c. The first captures the number of pairs a component differs and the other the number of pairs where the component is present. For each of the $n(n-1)/2$ pairs of samples in the family, it compares each of the 12 components. If a component c is present in both samples and is similar, it increments both counters for the component; if present in only one sample, the counters are not modified; and if the component is absent in both, it increments C_d^c. Once all pairs of samples have been analyzed, counters are normalized by dividing them by the number of pairs. For each component, if $C_p \geq t$ and $C_d \geq t$, the component is deemed *similar* (present and consistent

[6] For the Other Sections component, we sort the sections according to their offset, concatenate their raw bytes, and compute the SHA256 of the resulting buffer.

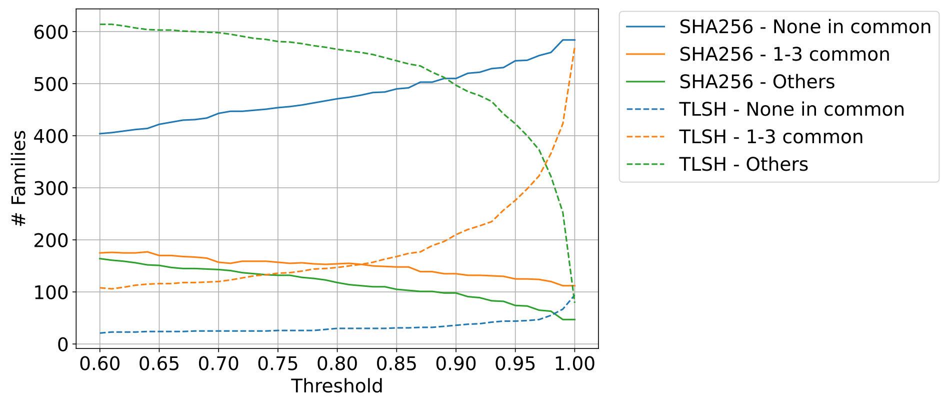

Fig. 1: Number of families with none, 1-3 or more than 3 common components by varying the threshold, and using SHA256 or TLSH.

in most samples), if $C_p < t$ and $C_d < t$, the component is present with varying values and is classified as *different*, and if $C_p < t$ and $C_d \geq t$, the component is often *missing* and thus should be ignored as there is not enough information.

Threshold selection. Requiring a component to be similar across all pairs of samples (i.e., t=1.0) is too strict, e.g., some samples could have wrong family labels. Instead, we evaluate lower threshold values that allow a fraction of pairs to differ. Figure 1 presents the number of families with none, few (1-3), or many (>3) common components while varying the threshold t for both SHA256 (solid lines) and TLSH (dashed lines). The results show clear differences between both pairwise similarity functions. Using SHA256, raising the threshold increases the number of families with no similar components gradually from 405 at t=0.6 to 510 at 0.9, while the number of common components gradually decreases in opposite fashion. In contrast, using TLSH the number of families with no common components remains fairly constant around 20 and only starts to increase at high thresholds. Our manual analysis shows that using TLSH is problematic with small data components (i.e., PE headers in Metadata) where component similarity is determined even when important fields differ. Thus, we conclude that using the stricter crypto hash is more appropriate to analyze polymorphism. While changing the threshold of the crypto hash affects a relatively small subset of families, too low a threshold reduces confidence that components are truly shared, while too high a threshold increases the risk of discarding slightly different but essentially similar data. For the experiments in Section 5 we use the SHA256 pairwise similarity with a threshold of 0.75 which we have observed to strike a good balance by tolerating small differences without overlooking meaningful commonalities.

4 Cross-Component Analysis

Before proceeding to the analysis of individual components, we analyze different sources of variability that are not specific to a component but may instead affect the entire PE file structure: file truncation and packing.

4.1 Truncation

Millions of malware samples are collected and shared daily by various companies and services. Some are retrieved directly from filesystems by endpoint protection systems, while others come from emails, compressed archives, network traffic, or memory. Additionally, files may be pre-processed by tools like unpackers. During this collection and transformation process, a sample might become truncated, either intentionally or unintentionally. Truncated samples can be identified because their expected size (i.e., the sum of the start offset and size of the last section) is larger than the real size of the file on disk. One caveat is that if a sample has an overlay and the truncation only affects the overlay, then truncation cannot be detected as the overlay is not described in the PE headers. Among the 68,683 samples in the three datasets, 2,504 (3.6%) are truncated. Of those, the vast majority (2,486) belong to the Balanced dataset and 18 come from MOTIF, with Malicia having none. Truncated samples are distributed among 42.9% (320/746) of the families, showing that truncation is indeed a widespread phenomenon. We found 1.9% (14/746) families where more than half of the samples are truncated. The families with the majority of truncated samples are: *stone* (97%), *kuaizip* (92%), *ranumbot* (88%), *duote* (87%), and *spybot* (82%).

To measure the missing part, we define the *truncation ratio* as the quotient of the missing bytes (expected size minus real size) over the expected size. The mean truncation ratio is 50.2% (median 52.9%), indicating that, on average, more than half of the expected file size is missing in truncated files. Then, by analyzing which components are impacted by truncation, we found that the offset from which bytes begin to be missing always starts at least after the Section Table; thus, truncation affects sections, resources, certificates, and overlay.

Another aspect of the truncation is whether the truncated samples came from the same original sample (i.e., the only difference is the truncation size). Thus, for each family, we compared all the truncated samples by checking whether the smaller sample was once again a truncated version of the largest. This analysis revealed 77 (10.3%) families having at least 2 samples coming from the same original executable and truncated in different points. The extreme case is one family, *spybot*, where 80 truncated samples come from the same original file that has been truncated at different offsets.

While truncated files are rarely, if ever, mentioned in malware studies, their consequences are very important. For instance, truncated samples cannot be executed, and thus fail dynamic analysis. But also static signatures may not match, and even popular static analysis tools produce confusing results when run on truncated files. For example, the popular *pefile* [5] (a multi-platform Python module to parse PE files) erroneously reports an inexistent overlay on all truncated samples. Given that also VirusTotal uses pefile to parse PE executables, VT often reports truncated samples having a non-existent overlay.

We remove the truncated samples, as well as any families with less than 10 non-truncated samples, from the dataset. The dataset used in the remainder of this paper has **66,160** samples distributed in **743** families.

4.2 Packing

We adopt a broad definition of packing, encompassing all transformations applied to a PE file—such as encryption, compression, or encoding—that require a runtime component to recover the original data. While these transformations primarily aim to thwart static analysis, they can also generate vastly different executables by re-packing the same sample with different algorithms or encryption keys. Consequently, packing is often regarded as a key driver of polymorphism.

Accurately detecting packed samples is a challenging task and an active research area. To address this, we employ two state-of-the-art approaches. Signature-based tools are effective at detecting known off-the-shelf packers, offering low false positives (FPs). However, they suffer from false negatives (FNs) due to their inability to identify custom packers or all instances of off-the-shelf packers. To overcome these limitations, we incorporate an ML classifier capable of detecting both less common off-the-shelf packers and custom packing routines. However, the classifier may overestimate the number of packed samples. In summary, we use signature-based tools to establish a lower bound and the ML classifier to set an upper bound on packing presence in our dataset.

Signature-based tools. We use two publicly available signature-based packer detection tools: PackGenome [27] and Detect-it-Easy [20]. PackGenome generates YARA rules from dynamic traces of the unpacking routine collected during program execution. The authors released the code for generating YARA rules for 20 off-the-shelf "accessible" packers. These rules identify 24,134 (36.5%) samples as being packed. The most commonly reported packers are UPX (48.6% of detected samples), WinLicense (31.7%) and PECompact (9.8%). The popular open-source Detect It Easy (DiE) tool, whose output is included in the VT reports at the time of writing, identified 15,704 samples (23.7%) being packed. The top-3 packers identified are UPX (58.8% of detected samples), ASPack (12.3%) and VMProtect (9.5%).

ML classifier. We implement a packer classifier by leveraging the datasets and the feature classes proposed by Aghakhani et al. [6]. On their datasets, we used the same nine static feature classes (56,485 individual features) they extracted to train a Random Forest classifier that predicts whether a sample is packed or not. We used 10-fold cross-validation to train and test the classifier and obtained an overall F1-score of 0.96. When applied to our dataset, the classifier predicted 55,773/66,160 (84.3%) samples as packed.

In summary, signature detection tools identify 23.7%–36.5% samples as packed, while the ML classifier identifies instead a much higher 84%, as summarized in

	Samples	**Families**				
Component	Present	Present(0%)	Present(≥50%)	Present(100%)	Differs	OnlyDifference
DOS Header	66,160 (100.0%)	-	743 (100.0%)	743 (100.0%)	524 (70.5%)	-
DOS Stub	65,059 (98.3%)	3 (0.4%)	734 (98.8%)	655 (88.2%)	484 (65.1%)	-
Rich Header †	42,577 (64.4%)	94 (12.7%)	497 (66.9%)	168 (22.6%)	494 (66.5%)	-
COFF Header	66,160 (100.0%)	-	743 (100.0%)	743 (100.0%)	622 (83.7%)	-
Optional Header	66,160 (100.0%)	-	743 (100.0%)	743 (100.0%)	652 (87.8%)	1 (0.1%)
Data Directories †	66,148 (99.9%)	-	743 (100.0%)	741 (99.7%)	638 (85.9%)	-
Section Table	66,160 (100.0%)	-	743 (100.0%)	743 (100.0%)	642 (86.4%)	-
Entry Point Section	66,155 (99.9%)	-	743 (100.0%)	739 (99.5%)	637 (85.7%)	2 (0.3%)
Resource Section †	57,230 (86.5%)	27 (3.6%)	661 (89.0%)	385 (51.8%)	625 (84.1%)	1 (0.1%)
Other Sections †	64,734 (97.8%)	3 (0.4%)	731 (98.4%)	583 (78.5%)	636 (85.6%)	7 (0.9%)
Certificate Table †	11,257 (17.0%)	344 (46.3%)	108 (14.5%)	19 (2.6%)	188 (25.3%)	1 (0.1%)
Overlay †	31,096 (47.0%)	87 (11.7%)	319 (42.9%)	87 (11.7%)	493 (66.4%)	37 (5.0%)

Table 2: Component statistics. For each component, number of samples where the component is present, number of families where the component is present in none/half/all samples, number of families where the component differs, and number of families where the component is the only one with differences. A dash means zero in all columns. A dagger † indicates an optional component.

Table 3 in Appendix. It is likely that the real fraction lies somewhere in between, so we use these numbers as lower and upper bounds, respectively. We illustrate the differences between both packer detection approaches using *privateexeprotector*, which has been mislabeled as a malware family but is rather a commercial protector. When the dataset by Dambra et al. [15] was created, AVClass taxonomy did not yet classify it as a packer, leading to its mislabeling. Executables labeled *privateexeprotector* may belong to different malware families using the same protector. Of 100 samples, 13 are tagged as "Private EXE Protector" by signature-based tools, 2 as UPX, and 87 remain undetected, showing high FNs. In contrast, the ML classifier detects 99 samples as packed, demonstrating minimal FNs, although potentially introducing FPs.

Finally, we noticed that many families use different off-the-shelf packers, a quick shortcut to achieve polymorphism. The most extreme cases are 8 families that use at least 15 different packers. In fact, we computed the Pearson correlation between the number of different components in each family and the total number of unique off-the-shelf packers in that family. We obtained a moderate relationship (0.42, p-value = 6.0e−34), namely, more packers, greater difference.

5 Component Analysis

We now use *PEdiff* to understand in which parts of a PE file the differences among samples of the same family are located. Section 5.1 first examines how frequently each component exists. Then, Section 5.2 identifies similar and different components in families. Section 5.3 examines where their polymorphism is being introduced in individual components.

5.1 Component Presence

Table 2 summarizes the presence of individual components in the dataset. Most required components are present in all samples (DOS Header, COFF Header, Optional Header, Section Table). Optional components vary significantly in presence. Three are common: Data Directories (present in 99.9% of samples), Other Sections (97.8%), and Resource Section (86.5%). Less common optional components include the Rich Header (64.4%), Overlay (47.0%), and Certificate Table (17.0%). The presence of an optional component is not consistent across all samples in a family. For example, while 399 (53.7%) families have some signed sample and 656 (88.3%) families have a sample with an overlay, only 19 (2.6%) families have all samples signed and only 87 (11.7%) have an overlay in all samples. Similarly, there are 94 families (12.7%) where no sample has the Rich Header, 168 (22.6%) where all samples have it, but in the majority of families some samples, but not all, have it (87.3%).

5.2 Similar and Different Components

This section examines how many similar and different components there are in each family. Roughly two-thirds of the dataset, specifically 454 families, contain samples with *no* components in common, i.e., where all parts of the PE files are, at least partially, different. Among the remaining 289 families, 95 families contain largely similar files, with only one, two, or three different components and 157 families contains instead files that are all different *except* for 1–3 components.

We examine these three family groups separately. In most families, differences among samples span all PE file components, not just localized parts. This suggests malware authors did not achieve polymorphism by altering a few bytes or simply re-compiling (which would preserve sections like data, resources, and the Rich Header). For the 42,498 samples in this group, 40.1% are packed according to the signature-based tools, 84.7% according to the ML classifier, and 90.4% when combining both methods. Of the samples in the remaining 289 families (23,662 samples), 38.4% are packed according to the signature-based tools, 83.6% according to the ML classifier, and 90.4% using both methods. Despite slightly higher percentages in the first group, the difference is too small to attribute polymorphism to packing alone.

We focus on families with files sharing few common components. The most consistent components are the DOS Header and Stub, in 95 and 131 families respectively, and the Rich Header, in 21 families. In highly similar families, the most variable component, is the overlay (78 families), appearing over three times more frequently than others like other sections (22), resource (17) and entry point (14) section. Rare differences include headers and, in three families, only the Certificate Table.

5.3 Individual Component Polymorphism

So far, our analysis has been binary—components were either identical or different. In this section, we explore differences and similarities in greater detail. For example, over 5% of the families in our dataset contain samples that differ only in their overlay. What do these overlays look like? Are they entirely different, or do they contain only a few unique bytes? To address these questions, we examine each component individually.

Rich Header. The Rich Header is an optional and undocumented component, which can be useful to detect if two executables may come from the same project. Among the 494 (66.5%) families where the Rich Header is different, in 12 families the entry IDs are similar, and in 4 family also its counts meaning that most objects used to build the samples are common.

Entry Point Section. This component contains the very first instruction of the program and thus can be considered a code section. The SHA256 pairwise similarity identifies 41 (5.5%) families where this component does not change at all and another 65 (8.7%) families where this component is similar. For the remaining 637 (85.7%) families, we use the BinDiff similarity to compare their code, identifying an additional 7 families with similar code. For 37/637 (5.81%) families BinDiff failed to properly disassemble all the samples, mainly because the entry point address points to invalid code. Such behavior is a common evasion technique in malware, where a custom loader, often implemented via TLS callbacks, dynamically reconstructs the correct code during runtime.

Resource Section. Among the 625 (84.1%) families where the resource section differs, there is one family (*dostre*) where this component is the only difference. The difference is in the resource values, in particular, there is one specific Bitmap resource whose value keeps changing. We manually reversed the code and discovered that the malicious code's payload (extracted and executed at runtime) was encoded within that image. We also identify four families (*moarider*, *winloadsda*, *axespec*, *sohana*) where their only difference lies in the order of the resource names, three families (*blackshades*, *umbra*, *virfire*) for which the resources are the same, but the padding of the section differs, ten families where the difference is in the Resource Table but not in the resources identifiers, and another (*lolbot*) has the only difference in a single string that changes for all the samples. The observed variations in the samples are likely introduced intentionally to achieve hash-bursting and, subsequently, polymorphism.

Other Sections. This component includes default sections with a pre-defined purpose (e.g., `.idata`, `.bss`, .rdata). But it is possible to create custom sections as well. This is the case of packers, which usually create one section for accomodating the unpacked payload. For one family (*ezsoftwareupdater*) the only difference in the whole executable is a single 32-byte string in the .rdata section, containing hexadecimal characters, likely a unique identifier, and in another family (*stormattack*) the difference was also in the .rdata section where a few hexadecimal strings were changing.

Overlay. To measure how much hidden data has been added to samples with an overlay, we define the *excess ratio* as the quotient of the size on disk over the expected size (i.e., without the overlay). The median excess ratio is 1.73x, an additional 73% content over the expected length. However, the mean is 130x, because some samples contain a vast amount of additional data. The extreme case is a sample with an expected size of 2.56KB but an overlay of 454MB.

We removed the overlays from all samples in the dataset and re-computed their SHA256 hash (which we will now refer to as "no-overlay hash"). About one-fourth (18,315/66,160) of the samples share the same no-overlay hash with at least another sample, indicating that the overlay was the only difference between them. In particular, 13 families only contain samples with the same no-overlay hash, and 46 families have at least 75% of their samples with the same no-overlay hash. Interestingly, 156 no-overlay hashes are shared across different families, with one matching samples in 12 families. This hash corresponds to `7zS.sfx`, a template for 7-zip self-extracting archives. These archives are created by combining `7zS.sfx`, a configuration file, and a compressed archive, with the overlay holding the latter two. Another hash matches 6 families and corresponds to `Default.sfx`, used by WinRAR for self-extracting files. These cases highlight the frequent use of self-extracting archives in malware for distributing multiple files, and likely also for obfuscation as analyzing them requires inspecting the overlay's compressed data.

Four families had overlay differences that stem solely from strings. In three cases, the overlay was entirely ASCII text, while the fourth had one meaningless differing string. However, examining the overlay revealed strings resembling CA names. Despite the Certificate Table fields being set to 0 in the Data Directory, treating the overlay as a Certificate Table revealed valid PKCS#7 signatures.

Finally, we analyzed overlay content using DiE. In 43.67% of cases (13,537/31,096), no file type was detected. Among the rest, 6.08% (1,890/31,096) were archives (RAR, ZIP, 7ZIP), 8.87% (2,750/31,096) were PE files, and 18.17% (5,650/31,096) contained ASCII text. Of these, 7.88% (445/5,650) were valid Base* encodings with no recognizable file types when decoded.

Certificate Table. For the 11,257 samples (17.0%) with a Certificate Table, we extracted the Authentihash and its hash algorithm. The Authentihash is computed from the file's content at signing, excluding the `Checksum` and `Certificate Table Data Directory`. We failed to extract a signature for 1,044 samples due to truncation or corruption (e.g., incorrect offset). For the remaining 10,213, a mismatched Authentihash indicated modification or unrelated signatures in 1,845 samples across 211 families. Thus, 8,368 samples (12.6%) had valid signatures when signed, though some may now be invalid (e.g., revoked). Among these, 1,098 samples had an overlay after the Certificate Table.

We computed the Authentihash for all samples using SHA256, resulting in 63,404 unique values, with 492 shared by multiple samples. Samples with identical Authentihash must belong to the same program (family and version), differing only in their Certificate Table and checksum. For instance, all the samples in the *amigo* family share the same Authentihash, checksum, and a chain of 5 certificates, indicating polymorphism in hidden parts of the Certificate Table.

Other Components. Of the remaining six components (DOS header, COFF and Optional Header, DOS Stub, Section Table, Data Directory), all except the DOS Stub have predefined structures in the PE format but can still be manipulated for polymorphism. For example, in *bebloh*, the only difference is the Optional Header's version values, while the code and data are identical. In *lebreat*, standard UPX section names (.upx0, .upx1, .rsrc) are replaced with random strings, though the content remains unchanged. We also analyzed the COFF Header's creation timestamp, which can be faked. It differs in 622 (83.7%) families, but 130 (11.3%) families share the same timestamp in over 75% of comparisons. Interestingly, *obit* has the same timestamp across all samples (Saturday, June 1, 2019 5:56:28 AM), likely fabricated, as the content differs.

5.4 File Infectors

File infectors, or viruses, infect benign executables with malicious code, creating samples with a combination of malicious and benign content. Furthermore, file infector families tend to be highly polymorphic since a single sample may infect many executables stored in the compromised host.

We investigated file infectors on our dataset using a combination of static and dynamic analysis. We first used AVClass [42] to obtain tags for all samples in the dataset. Using the tags, we identified 70 *likely-virus* families where the CLASS:virus tag appeared in more than half of the samples. For each of these families, we randomly selected 5 samples, dynamically executed them in a virtual machine (VM), and identified those that modified executables that already existed in the VM prior to the execution, i.e., the same filepath in the VM pointing to an executable file had different hashes before and after the execution. For those samples, we used *PEdiff* to examine the component differences between the original executable and the modified one produced during the execution.

Using this approach we identified 20 virus families. Of those, 16 are prepender viruses where the PE executable contains the malicious code and the infected (benign) executable is in the overlay: *lamer*, *induc*, *neshta*, *shodi*, *sinau*, *sivis*, *soulclose*, *xiaobaminer*, *memery*, *pidgeon*, *detroie*, *gogo*, *lmir*, *stihat*, *xolxo*, *xorer*. For all those 16 families, our analysis identifies the overlay as a component that changes. For two families (*gogo*, *soulclose*) the overlay is the only component that changes, i.e., the malicious executable has no polymorphism itself, but obtains it from the infected executable in the overlay. In the other 14 families, polymorphism is also added to other components. The remaining 4 families are appender viruses. Two of these (*expiro*, *wlksm*) extend one of the sections of the infected executable with the malicious code. The other two families (*triusor*, *wapomi*) add new sections at the end of the infected executable with malicious code. For all these four families, our analysis outputs that no component is similar across the family samples.

6 Related Work

Polymorphism. Malware achieves polymorphism by employing obfuscation techniques such as dead-code insertion, register reassignment, and instruction substitution [51]. This behavior renders static detection methods ineffective [7].

Therefore, prior works mainly focus on behavioral analysis to detect polymorphic malware: by using behavior-aware hidden Markov models [45], employing a mixed approach between static and dynamic analysis [35] or using an application-level emulator to perform flowgraph matching [13].

Code similarity. Some approaches compare executable files by examining the similarity of the disassembled code [18, 21, 22, 29]. These approaches may diff two versions of the same program [18, 29], search for similar programs in a repository [21], or group similar executables [22]. We leverage BinDiff [18] as a representative of this class to identify code similarity within a family.

Fuzzy hashes (or similarity hashes) compare file similarity at the raw byte level by generating digests that remain close in distance space for similar inputs. Various fuzzy hashes exist, including *SSDEEP* [26], *TLSH* [34], *SDHASH* [40], and *MRSH-v2* [12]. Prior research applies fuzzy hashes to forensic analysis [41], malware detection [31, 32, 43], and clustering [8, 9, 44, 48], while others evaluate their effectiveness [11, 36]. Instead, we propose a fine-grained structural comparison across 12 PE file components to pinpoint byte-level differences. As part of our approach, we use TLSH to implement pairwise comparison of the values of a component across two executables. We observe high volatility when aggregating TLSH comparisons for family-wise similarity due to few, but significant, differences in small components such as PE headers.

Malware clustering. A wealth of prior work has focused on grouping malware samples into families [10, 23, 37, 39]. Each family cluster typically contains executables from the same malicious program, which may include different versions or polymorphic variants of a version. Malware clustering methods may use similarities in system calls [10], network traffic [37, 39], raw bytes [23], or disassembled code [22]. We use three malware datasets where samples are labeled with their family and thus already clustered into families.

Malware lineage. Lineage methods classify malware family samples, identifying polymorphic variants and tracing their evolution through phylogenetic trees [14, 19, 24, 28, 46]. Instead of detecting identical versions, our study examines tactics employed to create polymorphic variants.

7 Final Remarks

A Complex Picture. Our study aimed to identify the main causes of polymorphism and assess their prevalence across a large dataset of malware families. Through our experiments, we identified several causes, summarized in the following section. More importantly, we found that a single factor is rarely sufficient to explain the diversity of samples within a family. In fact, only 12.8% of the families (95 out of 743) exhibited polymorphism due to a single cause. For the remaining 87.2%, polymorphism arose from multiple overlapping factors. This is not a failure but a key finding, highlighting that attributing polymorphism solely to repacking is an oversimplification. It also suggests that no holistic solution exists to address the problem. For example, while removing or normalizing certain components may help in comparing samples, any approach addressing only one or a few causes will have limited success in explaining and mitigating the dissimilarities within a family.

Truncation. While truncated files are rarely, if ever, mentioned in malware studies, we observed that 3.6% of the files in the initial datasets are truncated, with 99.3% coming from the balanced dataset by Dambra et al. Since that dataset was collected from the VirusTotal file feed, a similar ratio of truncated PE executables might affect other studies using the VT feed [49]. Truncated samples are distributed among nearly half of the dataset families, indicating this is not an issue specific to some families but likely a common error that occurs during sample collection. Truncated samples pollute malware feeds and waste resources such as storage and sandbox time if they are queued for execution. Therefore, we suggest filtering out these samples, as we did, to avoid biasing the results.

Overlays. Similar to truncation, the impact and role of overlays are rarely mentioned in malware studies. However, our experiments show that they are extremely prevalent, affecting a stunning 47.0% of all samples in our dataset, being the most prevalent cause of polymorphism that we find. These overlays often contain a considerable amount of data, on average over a hundred times larger than the main executable alone. Despite this, previous works sometimes purposefully excluded overlays when extracting features for static analysis [38]. This is fine if the overlay contains useless data simply added to achieve polymorphism, but our analysis shows that this is not the case: 6.1% of the overlay data are compressed archives, and 8.9% are PE files.

Packing. Packing is a pervasive phenomenon in our dataset: while it is difficult to measure with precision, it might affect between 40% to 90% of our samples. This is not surprising since it is one of the most effective methods to counter static analysis. However, one would expect a significant difference in the components between the families where packing was most prevalent, but the distributions of packed samples and the negligible correlations we found did not confirm this expectation. We also discover that malware authors, in a trivial but effective way, achieve high polymorphism by using many different packers.

Other polymorphism. Beyond packing, our study reveals that malware families introduce polymorphism into a variety of components. Among others, we observe families that modify PE headers to generate polymorphic variants such as *bebloh* that varies the version fields in the Optional Header. We also observe families that reorder resources without modifying them (e.g. *moarider*, *winloadsda*), introduce random bytes in the padding (*blackshades*, *umbra*, *virfire*), and introduce hidden data in the certificate table (*amigo*). There are also families whose differences are limited to some specific strings (e.g., *lolbot*). The range of techniques we observe shows that a structural analysis of the PE file format is a powerful tool for analyzing the reasons behind malware family polymorphism.

Dataset limitations. Our analysis is limited by the datasets used. One issue is the quality of family labels. Despite dataset authors' efforts to refine labeling (e.g., identifying aliases and generic tokens), we found some errors such as *priva-*

teexeprotector being considered a family. Also, the MOTIF and Malicia datasets are highly imbalanced, with few families having more than 10 samples. Still, our use of three datasets should help ameliorate selection bias.

Conclusions. Our large-scale analysis of 743 malware families offers a comprehensive understanding of the factors driving polymorphism in malware. The study reveals that in about 90% of cases, polymorphism results from multiple overlapping factors, rather than a single cause. This complexity underscores the inadequacy of simplistic solutions, such as attributing polymorphism solely to repacking, and highlights the need for multifaceted approaches.

Acknowledgements

This work was partially funded by two government grants managed by the French National Research Agency with references: "ANR-22-PECY-0007" and "ANR-23-IAS4-0001". Partial support was also provided by the Spanish Government MCIN/AEI/ 10.13039/501100011033/ through grants TED2021-132464B-I00 (PRODIGY) and PID2022-142290OB-I00 (ESPADA). The above grants are co-funded by European Union ESF, EIE, and NextGeneration funds.

References

1. Find malware detection names for Microsoft Defender for Endpoint. `https://learn.microsoft.com/en-us/microsoft-365/security/intelligence/malware-naming` (Accessed August 18, 2025)
2. Hash and family of each sample. `https://raw.githubusercontent.com/eurecom-s3/DecodingMLSecretsOfWindowsMalwareClassification/main/dataset/malware` (Accessed August 18, 2025)
3. MOTIF Dataset. `https://github.com/boozallen/MOTIF` (Accessed August 18, 2025)
4. PEdiff. `https://github.com/im-overlord04/PEDiff` (Accessed August 18, 2025)
5. PEfile. `https://github.com/erocarrera/pefile` (Accessed August 18, 2025)
6. Aghakhani, H., Gritti, F., Mecca, F., Lindorfer, M., Ortolani, S., Balzarotti, D., Vigna, G., Kruegel, C.: When malware is packin'heat; limits of machine learning classifiers based on static analysis features. In: Network and Distributed Systems Security Symposium (2020)
7. Arfeen, A., Khan, Z.A., Uddin, R., Ahsan, U.: Toward accurate and intelligent detection of malware. Concurrency and Computation: Practice and Experience **34**(4), e6652 (2022)
8. Azab, A., Layton, R., Alazab, M., Oliver, J.: Mining malware to detect variants. In: Cybercrime and Trustworthy Computing Conference (2014)
9. Bak, M., Papp, D., Tamás, C., Buttyán, L.: Clustering iot malware based on binary similarity. In: IEEE/IFIP Network Operations and Management Symposium (2020)
10. Bayer, U., Comparetti, P.M., Hlauschek, C., Kruegel, C., Kirda, E.: Scalable, Behavior-Based Malware Clustering. In: Network and Distributed System Security Symposium (2009)

11. Botacin, M., Moia, V.H.G., Ceschin, F., Henriques, M.A.A., Grégio, A.: Understanding uses and misuses of similarity hashing functions for malware detection and family clustering in actual scenarios. Forensic Science International: Digital Investigation **38**, 301220 (2021)
12. Breitinger, F., Baier, H.: Similarity preserving hashing: Eligible properties and a new algorithm mrsh-v2. In: International Conference on Digital Forensics and Cyber Crime (2013)
13. Cesare, S., Xiang, Y., Zhou, W.: Malwise—an effective and efficient classification system for packed and polymorphic malware. IEEE Transactions on Computers **62**(6), 1193–1206 (2012)
14. Cozzi, E., Vervier, P.A., Dell'Amico, M., Shen, Y., Bilge, L., Balzarotti, D.: The tangled genealogy of iot malware. In: Annual Computer Security Applications Conference (2020)
15. Dambra, S., Han, Y., Aonzo, S., Kotzias, P., Vitale, A., Caballero, J., Balzarotti, D., Bilge, L.: Decoding the Secrets of Machine Learning in Malware Classification: A Deep Dive into Datasets, Feature Extraction, and Model Performance. In: ACM Conference on Computer and Communications Security. ACM (November 2023)
16. Drew, J., Moore, T., Hahsler, M.: Polymorphic malware detection using sequence classification methods. In: IEEE Security and Privacy Workshops (2016)
17. Egele, M., Woo, M., Chapman, P., Brumley, D.: Blanket execution: Dynamic similarity testing for program binaries and components. In: USENIX Security Symposium (2014)
18. Google: BinDiff. `https://github.com/google/bindiff` (Accessed August 18, 2025)
19. Haq, I.U., Chica, S., Caballero, J., Jha, S.: Malware lineage in the wild. Computers & Security **78**, 347–363 (2018)
20. horsicq: Detect It Easy. `https://github.com/horsicq/Detect-It-Easy`, [Online; August 18, 2025]
21. Hu, X., Chiueh, T., Shin, K.G.: Large-scale Malware Indexing Using Function-call Graphs. In: ACM Conference on Computer and Communications Security (2009)
22. Hu, X., Shin, K.G., Bhatkar, S., Griffin, K.: MutantX-S: Scalable Malware Clustering Based on Static Features. In: USENIX Annual Technical Conference (2013)
23. Jang, J., Brumley, D., Venkataraman, S.: Bitshred: feature hashing malware for scalable triage and semantic analysis. In: ACM conference on Computer and Communications Security (2011)
24. Jang, J., Woo, M., Brumley, D.: Towards Automatic Software Lineage Inference. In: USENIX Security Symposium (2013)
25. Joyce, R.J., Amlani, D., Nicholas, C., Raff, E.: MOTIF: A large malware reference dataset with ground truth family labels. In: Workshop on Artificial Intelligence for Cyber Security (2022)
26. Kornblum, J.: Identifying almost identical files using context triggered piecewise hashing. Digital investigation **3** (2006)
27. Li, S., Ming, J., Qiu, P., Chen, Q., Liu, L., Bao, H., Wang, Q., Jia, C.: Packgenome: Automatically generating robust yara rules for accurate malware packer detection. In: ACM SIGSAC Conference on Computer and Communications Security (2023)
28. Lindorfer, M., Di Federico, A., Maggi, F., Comparetti, P.M., Zanero, S.: Lines of Malicious Code: Insights into the Malicious Software Industry. In: Annual Computer Security Applications Conference (2012)
29. Liu, B., Huo, W., Zhang, C., Li, W., Li, F., Piao, A., Zou, W.: αDiff: Cross-version Binary Code Similarity Detection with DNN. In: ACM/IEEE International Conference on Automated Software Engineering (2018)

30. Microsoft: PE format (2023), `https://learn.microsoft.com/en-us/windows/win32/debug/pe-format`
31. Naik, N., Jenkins, P., Savage, N.: A ransomware detection method using fuzzy hashing for mitigating the risk of occlusion of information systems. In: International Symposium on Systems Engineering (2019)
32. Naik, N., Jenkins, P., Savage, N., Yang, L., Naik, K., Song, J., Boongoen, T., Iam-On, N.: Fuzzy hashing aided enhanced yara rules for malware triaging. In: IEEE Symposium Series on Computational Intelligence (2020)
33. Nappa, A., Rafique, M.Z., Caballero, J.: The MALICIA Dataset: Identification and Analysis of Drive-by Download Operations. International Journal of Information Security **14**(1), 15–33 (February 2015)
34. Oliver, J., Cheng, C., Chen, Y.: Tlsh–a locality sensitive hash. In: Cybercrime and Trustworthy Computing Workshop (2013)
35. Osorio, F.C.C., Qiu, H., Arrott, A.: Segmented sandboxing-a novel approach to malware polymorphism detection. In: International Conference on Malicious and Unwanted Software (2015)
36. Pagani, F., Dell'Amico, M., Balzarotti, D.: Beyond precision and recall: understanding uses (and misuses) of similarity hashes in binary analysis. In: ACM Conference on Data and Application Security and Privacy (2018)
37. Perdisci, R., Lee, W., Feamster, N.: Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces. In: USENIX Symposium on Networked Systems Design and Implementation (2010)
38. Quiring, E., Pirch, L., Reimsbach, M., Arp, D., Rieck, K.: Against all odds: Winning the defense challenge in an evasion competition with diversification. Tech. rep. (2020)
39. Rafique, M.Z., Caballero, J.: FIRMA: Malware Clustering and Network Signature Generation with Mixed Network Behaviors. In: Symposium on Research in Attacks, Intrusions and Defenses (2013)
40. Roussev, V.: Data fingerprinting with similarity digests. In: IFIP International Conference on Digital Forensics (2010)
41. Roussev, V., Quates, C.: Content triage with similarity digests: The m57 case study. Digital Investigation **9**, S60–S68 (2012)
42. Sebastián, M., Rivera, R., Kotzias, P., Caballero, J.: AVClass: A Tool for Massive Malware Labeling. In: International Symposium on Research in Attacks, Intrusions, and Defenses (2016)
43. Seo, K., Lim, K., Choi, J., Chang, K., Lee, S.: Detecting similar files based on hash and statistical analysis for digital forensic investigation. In: International Conference on Computer Science and Its Applications (2009)
44. Shiel, I., O'Shaughnessy, S.: Improving file-level fuzzy hashes for malware variant classification. Digital Investigation **28**, S88–S94 (2019)
45. Tajoddin, A., Jalili, S.: Hm 3 ald: Polymorphic malware detection using program behavior-aware hidden markov model. Applied Sciences **8**(7), 1044 (2018)
46. Tam, K., Feizollah, A., Anuar, N.B., Salleh, R., Cavallaro, L.: The evolution of android malware and android analysis techniques. ACM Computing Surveys (CSUR) **49**(4), 1–41 (2017)
47. Upchurch, J., Zhou, X.: Variant: a malware similarity testing framework. In: International Conference on Malicious and Unwanted Software (2015)
48. Upchurch, J., Zhou, X.: Malware provenance: code reuse detection in malicious software at scale. In: International Conference on Malicious and Unwanted Software (2016)

49. van Liebergen, K., Caballero, J., Kotzias, P., Gates, C.: A Deep Dive into the VirusTotal File Feed. In: Conference on Detection of Intrusions and Malware & Vulnerability Assessment (2023)
50. Webster, G.D., Kolosnjaji, B., von Pentz, C., Kirsch, J., Hanif, Z.D., Zarras, A., Eckert, C.: Finding the needle: A study of the pe32 rich header and respective malware triage. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (2017)
51. You, I., Yim, K.: Malware obfuscation techniques: A brief survey. In: International Conference on Broadband, Wireless Computing, Communication and Applications (2010)

A Appendix

	Samples	**Families**	
Factor	Present	Present(>0%)	Present(100%)
Packed (DiE)	15,704 (23.7%)	527 (70.9%)	20 (3.0%)
Packed (PackG)	24,134 (36.5%)	580 (78.1%)	45 (6.1%)
Packed (ML)	55,773 (84.3%)	726 (97.7%)	283 (38.1%)
Packed (All)	59,265 (89.6%)	731 (98.4%)	354 (47.6%)

Table 3: Packing prevalence in terms of packed samples and number of families with some/all packed samples.

Algorithm 1 Determine Component Status: Similar, Different, or Missing

```
Require: Array of samples $S$ belonging to family $F$, Component $c$, Threshold $t$
 1: Initialize $C_p \leftarrow 0$
 2: Initialize $C_d \leftarrow 0$
 3: $NS \leftarrow |S|$, number of samples in S
 4: $P \leftarrow$ all combinations of $S$
 5: $NP \leftarrow \frac{NS(NS-1)}{2}$, number of samples combinations and cardinality of P

 6: for all $(S_x, S_y) \in P$ do
 7:     if $c \in S_x$ and $c \in S_y$ and $S_x[c] = S_y[c]$ then
 8:         $C_p \leftarrow C_p + 1$
 9:         $C_d \leftarrow C_d + 1$
10:     else if $c \notin S_x$ and $c \notin S_y$ then
11:         $C_d \leftarrow C_d + 1$
12:     end if
13: end for
14: $C_p \leftarrow \frac{C_p}{NP}$
15: $C_d \leftarrow \frac{C_d}{NP}$

16: if $C_p \geq t$ and $C_d \geq t$ then
17:     return Similar
18: else if $C_p < t$ and $C_d < t$ then
19:     return Different
20: else
21:     return Missing
22: end if
```