The Tangled Genealogy of IoT Malware

Emanuele Cozzi
emanuele.cozzi@eurecom.fr
EURECOM
Sophia Antipolis, France

Pierre-Antoine Vervier
France

Matteo Dell'Amico
della@linux.it
France

Yun Shen
yun.shen@nortonlifelock.com
NortonLifeLock, Inc.
Reading, United Kingdom

Leyla Bilge
leylya.yumer@nortonlifelock.com
NortonLifeLock, Inc.
Sophia Antipolis, France

Davide Balzarotti
davide.balzarotti@eurecom.fr
EURECOM
Sophia Antipolis, France

ABSTRACT

The recent emergence of consumer off-the-shelf embedded (IoT) devices and the rise of large-scale IoT botnets has dramatically increased the volume and sophistication of Linux malware observed in the wild. The security community has put a lot of effort to document these threats but analysts mostly rely on manual work, which makes it difficult to scale and hard to regularly maintain. Moreover, the vast amount of code reuse that characterizes IoT malware calls for an automated approach to detect similarities and identify the phylogenetic tree of each family.

In this paper we present the largest measurement of IoT malware to date. We systematically reconstruct – through the use of binary code similarity – the lineage of IoT malware families, and track their relationships, evolution, and variants. We apply our technique on a dataset of more than 93k samples submitted to VirusTotal over a period of 3.5 years. We discuss the findings of our analysis and present several case studies to highlight the tangled relationships of IoT malware.

CCS CONCEPTS

• **Security and privacy** → *Software and application security*; **Malware and its mitigation**.

KEYWORDS

Malware, IoT, Classification, Measurement, Lineage

ACM Reference Format:
Emanuele Cozzi, Pierre-Antoine Vervier, Matteo Dell'Amico, Yun Shen, Leyla Bilge, and Davide Balzarotti. 2020. The Tangled Genealogy of IoT Malware. In *Annual Computer Security Applications Conference (ACSAC 2020), December 7–11, 2020, Austin, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3427228.3427256

1 INTRODUCTION

Over the last few years we have witnessed an increase in both the volume and sophistication of malware targeting IoT systems. Traditional botnets and DDoS tools now cohabit with crypto-mining, spyware, ransomware, and targeted samples designed to conduct cyber espionage. To make things worse, the public availability of the source code associated with some of the main IoT malware families have paved the way for myriads of variants and tangled relationships of similarities and code reuse. To make sense of this complex evolution, the security community has devoted a considerable effort to analyze and document these emerging threats, mostly through a number of blog posts and the definitions of Indicators of Compromise [7, 8, 36, 44]. However, while the insights gained from these reports are invaluable, they provide a very scattered view of the IoT malware ecosystem.

On the academic side, Cozzi *et al.* [12] provided the first large-scale study of Linux malware by relying on a combination of static and dynamic analyses. The authors studied the behavior of 10K samples collected between November 2016 and November 2017, with the goal of documenting the sophistication of IoT malware (in terms of persistence mechanisms, anti-analysis tricks, packing, etc.). Antonakakis *et al.* [3] instead dissected the *Mirai* botnet and provided a thorough investigation into its operations, while Pa *et al.* [35] and Vervier *et al.* [45] used IoT honeypots to measure the infection and monetization mechanisms of IoT malware.

Despite this effort, little is still known about the dynamics behind the emergence of new malware strains and today IoT malware is still classified based on the labels assigned by AV vendors. Unfortunately, these labels are often very coarse-grained, and therefore unable to capture the continuous evolution and code sharing that characterize IoT malware. For instance, it is still unclear how many variants of the *Mirai* botnet have been observed in the wild, and what makes each group different from the others. We also have a poor understanding of the inner relationships that link together popular families, such as the *Mirai* and *Gafgyt* botnets and the infamous *VPNFilter* malware.

This paper aims at filling this gap by proposing a systematic way to compare IoT malware samples and display their evolution in a set of easy-to-understand lineage graphs. While there exists a large corpus of works that focused on the clustering of traditional malware [5, 6, 25, 29, 38] and exploring their lineage [15, 24, 26, 28, 31, 33] proving the complexity of these problems, in this paper we show that the peculiarities of IoT malware require the adoption of

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

ACM ISBN 978-1-4503-8858-0/20/12…$15.00
https://doi.org/10.1145/3427228.3427256

customized techniques. On the other hand, to our advantage, the current number of samples and the general lack of code obfuscation make possible, for the first time, to draw a complete picture that covers the entire ecosystem of IoT malware.

Our main contribution is twofold. First, we present an approach to reconstruct the lineage of IoT malware families and track their evolution. This technique allows identifying various variants of each family and also the intra-family relationships that occur due to the code-reuse among them. Second, we report on the insights gained by applying our approach on the largest dataset of IoT malware ever assembled to date, which include all malicious samples collected by VirusTotal between January 2015 and August 2018[1].

Our lineage graphs enabled us to quickly discover over a hundred mislabeled samples and to assign the proper name to those for which AV products did not reach a consensus. Overall, we identified and validated over 200 variants in the top families alone, we show the speed at which new variants were released, and we measured for how long new samples belonging to each variant appeared on VirusTotal. By looking at changes in the functions, we also identify a constant evolution of thousands of small variations within each malware variant. Finally, our experiments also emphasize how the frequent code reuse and the tangled relationship among all IoT families complicate the problems of assigning a name to a given sample, and to clearly separate the end of a family and the beginning of another.

We make the full dataset and the raw results available to researchers [2]. We also share the high resolution figures of the lineage graphs made by architecture for ease of exploration.

1.1 Why this Study Matters

IoT malware is an important emerging phenomenon [35], not just because of its recent development but also because IoT devices might not be able to run anti-malware solutions comparable to those we use today to protect desktop computers. However, to be able to design new solutions, it is important for the security community to precisely understand the characteristics of the current threat landscape. This need prompted researchers to conduct several measurement studies, focused for instance on the impact of the *Mirai* botnet [3] or on the techniques used by Linux-based malicious samples [12].

This work follows the same direction, but it is over one order of magnitude larger than previous studies and includes *all* malicious samples submitted to VirusTotal over a period of 3.5 years. A consequence of the scale of the measurement is that the manual analysis used in previous studies had to be replaced with fully automated comparison and clustering techniques.

Our findings are not just curiosities, but carry important consequences for future research in this field. For example, static analysis was the preferred choice for program analysis, until researchers showed that the widespread use of packing and obfuscation made it unsuitable in the malware domain [34]. Our work shows that this is not yet the case in the IoT space, and that today static code analysis provides more accurate results than looking at dynamic sandbox reports or static features. The fragmentation of IoT families also casts some doubts on the ability of AV labels to characterize the complex and tangled evolution of IoT samples.

Finally, while not our main contribution, our work also reports on the largest clustering experiments conducted to date on dynamic features extracted from malicious samples [5, 6, 25].

2 DATASET

To study the genealogy of IoT malware, our first goal was to collect a large and representative dataset of malware samples. For this purpose, we downloaded all ELF binaries that have been submitted to VirusTotal [2] over a period of almost four years (from January 2015 to August 2018) and that had been flagged as malicious by at least five anti-virus (AV) vendors. Since our goal is to analyze malware that targets IoT devices, we purposely discarded all Android applications and shared libraries. Furthermore, we also removed samples compiled for the Intel and AMD architectures because it is very difficult to distinguish the binaries for embedded devices from the binaries for Linux desktop systems. This selection criteria resulted in a dataset of 93,652 samples, one order of magnitude larger than any other study conducted on Linux-based malware. As a comparison, the largest measurement study to date was performed on 10,548 Linux binaries [12], of which a considerable fraction (64.56%) were malware targeting x86 desktop computers. Moreover the purpose of this dataset was to study the general behavior of modern Linux malware and not the tangled relationships between them.

We could have easily extended our dataset to Linux malware for desktops and servers. On the other hand, we preferred to focus specifically on IoT malware, given their high infection rate on real devices and the variety of the underlying hardware architectures. This possibly requires platform customizations implemented as ad-hoc malware variants. Moreover, less known architectures are more likely to show those small bits which tend to be ignored on more comfortable and extensively studied counterparts e.g., x86.

Figure 1 shows the volume of samples in our dataset submitted to VirusTotal over the data collection period and the dramatic increase in the number of IoT malware samples after the outbreak of the infamous *Mirai* botnet in October 2016. Before that, the number of malicious IoT binaries was very low. For instance, only 363 of our 93K samples were observed in that period. This number progressively increased to reach an average of 7.8k new malicious binaries per month in 2018. This trend can be attributed to several factors, including the evolving IoT threat landscape [27, 42, 43, 45], the source code availability of several popular families [27], and the proliferation of IoT honeypots that allowed researchers to rapidly collect a large number of samples spreading in the wild [45].

Table 1 reports the compilation details of the samples in our dataset. The first two architectures, *ARM 32-bit* and *MIPS I*, account together for two thirds of all samples. This can be explained by the large popularity of these processor architectures for popular consumer IoT devices commonly targeted by these malware, such as home routers, IP cameras, printers, and NAS devices. Another interesting aspect is the fact that almost 95% of the ELF files in our dataset were statically linked. Additionally, as already noted by Cozzi et al. [12], a large fraction of them (roughly 50% in our dataset) have not been stripped from their symbols.

[1] As explained in Section 2, we included in our analysis only samples detected as malicious by at least five AV systems.

[2] Dataset and figures: *https://github.com/eurecom-s3/tangled_iot/*

Table 1: Breakdown of samples per architecture.

CPU Architecture	Samples No. (%)	Dynamically Linked Stripped	Unstripped	Total	Statically Linked Stripped	Unstripped	Total
ARM 32-bit	36,574 (39.05)	3,012	645	3,657	16,049	16,868	32,917
MIPS I	25,201 (26.91)	325	345	670	12,714	11,817	24,531
PowerPC 32-bit	10,916 (11.66)	100	258	358	5,180	5,378	10,558
SPARC	8,412 (8.98)	100	119	219	3,489	4,704	8,193
Hitachi SH	6,477 (6.92)	63	107	170	2,190	4,117	6,307
Motorola 68000	5,982 (6.39)	52	82	134	2,130	3,718	5,848
Tilera TILE-Gx	27 (0.03)	0	1	1	26	0	26
ARC International ARCompact	27 (0.03)	16	2	18	7	2	9
Interim Value tba	9 (0.01)	2	7	9	0	0	0
SPARC Version 9	8 (0.01)	1	6	7	1	0	1
PowerPC 64-bit	6 (0.01)	1	4	5	1	0	1
Others	13 (0.01)	4	8	12	1	0	1
Total	93,652	3,676	1,584	5,260	41,788	46,604	88,392

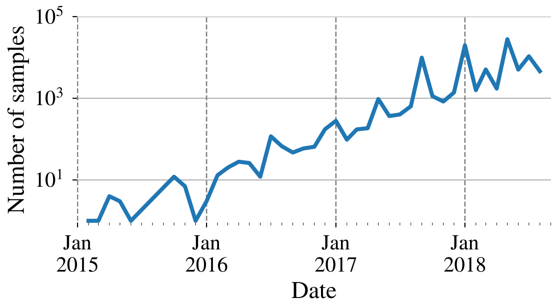

Figure 1: Number of samples in our dataset submitted to VirusTotal over time.

Table 2: Breakdown of the top 10 IoT malware families in our dataset.

Rank	Label (AVClass)	Samples No. (%)
1	Gafgyt	46,844 (50.02)
2	Mirai	33,480 (35.75)
3	Tsunami	3,364 (3.97)
4	Dnsamp	2,235 (3.59)
5	Hajime	1,685 (2.39)
6	Ddostf	840 (0.90)
7	Lightaidra	360 (0.38)
8	Pnscan	212 (0.23)
9	Skeeyah	178 (0.19)
10	VPNFilter	135 (0.14)
	Total	89,935 (96.03)
	Unlabelled	3,717 (3.97)

In addition to downloading the binaries, we also retrieved the VirusTotal reports. We then processed them with AVClass [41], a state-of-the-art technique that relies on the consensus among the AV vendors to determine the most likely family name attributed to malware samples. Table 2 lists the top ten AVClass labels, with *Gafgyt* and *Mirai* largely dominating the dataset. However, there is a long tail of families (90 in total) that contain only a small number of samples. Finally, it is interesting to note that AVClass was unable to find a consensus for a common family name for only 3.7K samples. While this might seem very small (especially compared with figures obtained on Windows malware), if we remove *Mirai* and *Gafgyt*, a common label was not found for one third of the remaining samples.

3 MALWARE LINEAGE GRAPH EXTRACTION

The field that studies the evolution of malware families and the way malware authors reuse code between families as well as between variants of the same family is known as *malware lineage.* Deriving an accurate lineage is a difficult task, which in previous studies has often been performed with help from manual analysis and over a small limited number of samples [24, 31]. However, given the scale of our dataset, we need to rely on a fully automated solution. The traditional approach for this purpose is to perform malware clustering based on static and dynamic features [15, 25, 28, 31]. When also the time dimension is combined in the analysis, clustering can help derive a complete timeline of malware evolution, also known as *phylogenetic tree* of the malware families.

A common and simple other way to do that would be to rely on AV labels, more oriented to only identify macro-families.

We, on the other hand, work towards a finer-grained classification that would enable us to study differences among sub-families and the overall intra-family evolution and relationships.

In our first attempt we decided to cluster samples based on a broad set of both static and dynamic features. This approach not only required a substantial amount of manual adjustments and validation, it also always resulted in noisy clusters. As feature-based clustering is often used in malware studies, we believe there is a value in reporting the reasons behind its failure. We thus provide a detailed analysis in Appendix A with the complete list of extracted features in Appendix B.

3.1 Code-based Clustering

We decided to resort to a more complex and time consuming solution based on code-level analysis and function similarity. The advantage is that code does not lie, and therefore can be used to precisely track both the evolution over time of a given family as well as the code reuse and functionalities borrowed among different families.

The main drawback of clustering based on code similarity is that the distance among two binaries is difficult to compute. Binary code similarity is still a very active research area [21], but tools that can scale to our dataset size are scarce and often in a prototype form.

Moreover, to be able to compare binaries, three important conditions must be satisfied: 1) each sample needs to be first properly unpacked, 2) it must be possible to correctly disassemble its code, and 3) it must be possible to separate the code of the application from the code of its libraries. The first two constitute major problems that had hindered similar experiments on Windows malware. However, IoT malware samples are still largely un-obfuscated and packers are the exception instead of the norm [12]. While this is a promising start, the third condition turns out to be a difficult issue (ironically this is the only one not causing problems for traditional Windows malware).

Figure 2 shows the workflow of our code-based clustering. The process is divided in three macro phases. Ⓐ First we process unstripped binaries and we analyze the symbols to locate library code in statically linked files. Ⓑ Then we perform an incremental clustering based on the code-level similarity, while propagating symbols to each new sample. Ⓒ Finally, we build the family graphs (one for each CPU architecture) and Ⓓ we use available symbols to pin samples and clusters to code snippets we were able to scrape from online code repositories to obtain more detailed understanding about the evolution of malware families.

Recall that our goal is not to provide a future-proof IoT malware analysis technique. We rather seek to identify a scalable approach that enables us to reconstruct the lineage for the 93K samples in our 3.5 year-long dataset so we can report on their genealogy. We thus take advantage of the current sophistication of IoT malware, which is currently rudimentary enough to enable code-based analysis, aware that malware authors could easily employ tricks to hinder such analysis in the future.

3.2 Symbols Extraction

IoT malware is often shipped statically linked. The fact that 88,392 samples out of 93,652 (94.3%) in our dataset are statically linked tend to confirm this assumption. This is most likely due to an effort to ensure the samples can run on devices with various system configurations. However, performing code similarity on statically linked binaries is useless, as two samples would be erroneously considered very similar simply because they might include the same *libc* library. Therefore, to be able to identify the relevant functions in such binaries, we first need to distinguish the user-defined code from the library code embedded in them. Unfortunately, when dealing with stripped binaries, this is still an open problem and the techniques proposed to date have large margins of errors, which are not suitable for our large-scale, unsupervised experiments.

We thus start our analysis by extracting symbols from unstripped binaries and leveraging them to add semantics to the disassembled code. Luckily, as depicted in Table 1, 53% of statically-linked and 30% of dynamically linked samples contain symbol information. We used IDA Pro to recognize functions and extract their names. We then use a simple heuristic to cut the binary in two. The idea is to locate some library code, and then simply consider everything that comes after library code as well. While it is possible for the linker to interleave application and library objects in the final executable, this would simply result in discarding part of the malware code from our analysis. However, this is not a common behavior, and lacking any better solution to handle this problem, this is a small price to pay to be able to compute binary similarity on our dataset.

We therefore built a database of symbols (symbols DB in Figure 2) extracted from different versions of *Glibc* and *uClibc* and use the database to find a "cut" that separates user from library code. After extracting the function symbols from unstripped ELF samples, we start scanning them linearly with respect to their offsets. We move a sliding window starting from the entry point function `_start` and define a cutting point as soon as all of the function names within that window have a positive match in the symbols DB. Using a window instead of a single function match avoids erroneous cases where a user function name may be wrongly interpreted as a library function. We experimentally set this window size to 2 and verified the reliability of this heuristic by manually analyzing 100 cases. Once the cutting point is identified, all symbols before this point are kept and the remaining ones are discarded.

We chose to operate only on *libc* variants for two reasons. First, because *libc* is always included by default by compilers into the final executable when producing statically linked files. Moreover, we observed that less than 2% of the dynamically linked samples in our dataset require other libraries on top of *libc*.

Finally, after removing the library code, we further filter out other special symbols, including `__start`, `_start` and architecture-dependent helpers like the `__aeabi_*` functions for ARM processors.

3.3 Binary Diffing and Symbol Propagation

Binary diffing constitutes the core of our approach as it enables us to assess the similarity between binaries at the code level. However, given the intrinsic differences at the (assembly) code level between binaries of different architectures, we decided to diff together only binaries compiled for the same architecture – therefore producing a different clustering graph, and a different malware lineage, for each architecture. While this choice largely reduces the number of possible comparisons, our datasets still contains up to 36,574 files per architecture (ARM 32-bit), making the computation of a full similarity matrix unfeasible.

To mitigate this problem we adopt Hierarchical Navigable Small World graphs (HNSW) [32], an efficient data structure for approximate nearest neighbor discovery in non-metric spaces, to overcome the time complexity and discover similarities in our dataset. The core idea that accelerates this and similar approaches [14, 17] is that items only get compared to neighbors of previously-discovered neighbors, drastically limiting the number of comparisons while still maintaining high accuracy. While adding files to the HNSW, our distance function will be called on a limited number of file pairs (on average, adding an element to the HNSW requires only 244 comparisons in our case) while still being able to link it to its most similar neighbors. We configured the HNSW algorithm to take advantage of parallelism and provide high-quality results as suggested by existing guidelines in the clustering literature [13].

We use *Diaphora* [1] to define our dissimilarity function for HNSW. This function is non-metric as the triangle inequality rule does not necessarily hold. However, in the following we will call it *distance function* without implying it is a proper metric. This has not consequences on the precision of our clustering, as the HNSW

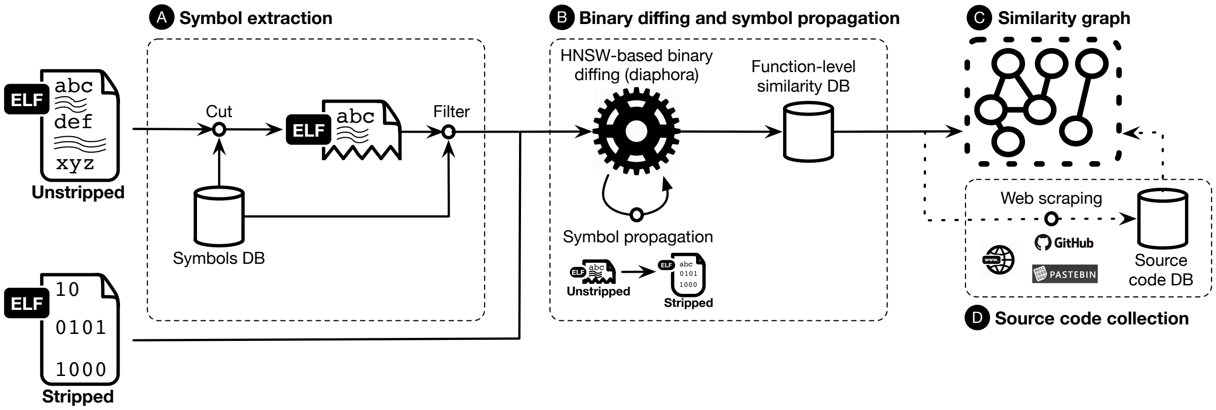

Figure 2: The workflow of our system.

algorithm is explicitly designed for non-metric spaces. One of the advantages of using Diaphora is that the tool works with all the architectures supported by IDA Pro, which covers 11 processors and 99.9% of the samples in our dataset, while other binary code similarity solutions recently proposed in academia handle only few architectures and do not provide publicly available implementations [21]. When two binaries are compared, Diaphora outputs a per-function similarity score ranging from 0 (no match) to 1 (perfect match). To aggregate individual function scores in a single distance function we experimented with different solutions based on the average, maximum, normalized average, and sum of the scores. We finally decided to count the number of functions with similarity greater than 0.5, which is the threshold suggested by Diaphora's authors to discard *unreliable* results. This has the advantage of providing a symmetric score (e.g., if the similarity of A to B is 4 then the similarity of B to A is also 4) that constantly increase as more and more matching functions are found among two binaries. For HNSW we then report the inverse of this count to translate the value into a distance (where higher values mean two samples are further apart and lower values mean they share more code).

Before running HNSW to perform pairwise comparison on the whole dataset, we unpacked 6,752 packed samples. Since they were all based on variations of UPX, we were able to easily do that by using a simple generic unpacker. We then add each sample to HNSW one by one, in two rounds, sorted by their first seen timestamp on VirusTotal (to simulate the way an analyst would proceed when collecting new samples over time).

In the first round we added all dynamically linked or unstripped samples, which account for 55% of the entire dataset. By relying on the symbols extracted in the previous phase, we only perform the binary diffing on the user-defined portion of the code, and omit comparisons on library code. In the second round we then added the statically linked stripped samples. Being without symbols, there is no direct way to distinguish user functions from library code. Attempts to recover debugging information from stripped binaries, such as with *Debin* [22], only target a limited set of CPU architectures.

We tackle this problem by leveraging the binary diffing itself to iteratively "propagate" symbols. When a function in a stripped sample has perfect similarity with an unstripped one, we label it with the same symbol. This methodology enables us to perform similarity analysis also for stripped samples, which would otherwise be discarded. However, this step comes with some limitations. While we are able to discard library functions we also potentially discard user functions that didn't match any function already in the graph. For instance, if two stripped statically linked samples share a function that is never observed in unstripped or dynamically linked binaries, this similarity would not be detected by our solution. We add the stripped samples to HNSW only after the unstripped ones have all been added to contain this problem as much as possible, but the probabilistic nature of HNSW can decrease this benefit as not all comparisons are computed for each sample. This means that our graph is an under-approximation of the perfect similarity graph (we can miss some edges that would link together different samples, but not create false connections) with over 18.7M one-to-one binary comparisons and 595,039 function symbols propagated from unstripped to stripped binaries.

3.4 Source Code Collection

The symbols extracted from unstripped malware and propagated in the similarity phase also helps us locate and collect snippets of source codes from online sources. In fact, the source code for many Linux-based IoT malware families has been leaked on open repositories hosting malicious packages ready to be compiled and deployed. This has resulted in a very active community of developers that cloned, reused, adapted, and often re-shared variations of existing code.

We took advantage of this to recognize open source and closed source families, split our dataset accordingly and, more importantly, to assign labels to groups of nodes in the similarity graph. While we also use AV labels for this purpose, those labels often correspond to generic family names, while online sources can help disambiguate specific variants within the same bigger family.

To locate examples of source code, we queried search engines with the list of user-defined function symbols extracted in the previous phase. We were able to find several matches on public services as GitHub or Pastebin, both for entire code bases (e.g., on GitHub) and for single source files (e.g., on Pastebin). Interestingly, on GitHub we found tens of repositories forked thousands of times (not necessarily for malicious purposes, as often security researchers also forked those repositories). Moreover, we found a Russian website hosting a repository regularly populated with several malware projects, exploits, and cross-compilation resources. From this source alone we were able to retrieve the code of 76 variants of *Gafgyt*, 50 variants of *Mirai*, 19 projects generically referred as "CnC Botnet" and "IRC Sources" (which resemble *Tsunami* variants) and a number of exploits for widely deployed router brands. Some variants contained changelog information that made us believe these projects had been collected from leaks and underground forums.

3.5 Phylogenetic Tree of IoT Malware

As a preamble to the function level similarity analysis of IoT malware we post-processed the sparse similarity graph G obtained by running HNSW and using the distance function as weight. Since we store in a database the detailed comparisons, the actual weight on the similarity graph can be tuned depending on the purpose of the analysis.

For instance, the analyst can use only best matches if the goal is to highlight perfect similarity (e.g., code reused as is) between two binaries, or a combination of best and partial matches if we want to capture more generic dependencies between two binaries, including minor variations and "evolutions" of the code.

Another problem with the similarity graph is that it contains a large number of edges, with many samples being variations (or simple recompilation) of the same family. Therefore, to make the output more readable and better emphasize the evolution lines, in our graphs we visualize the Minimum Spanning Tree (*MST*) G' of G that shows the path of minimum binary difference among all samples. This approach to cluster binaries is inspired by the works in clustering literature that are based on the minimum spanning tree (MST) of the pairwise distance matrix between elements [4, 11].

Furthermore, we observed that MSTs—which are in general used as an intermediate representation of the clustering structure—faithfully convey information about the relationships between items in our dataset which is not always preserved when converting the MST to a set of clusters. For this reason, we base our analysis on minimum spanning trees.

The tree can be further colored according to AV labels (to get an overview of the relationships among different families and spot erroneous labels assigned by AV engines) or to the closest source file we downloaded using the symbol names (thus leading to a more clear picture of the genealogy of a single malware family). In the next sections we will explore these two views and present a number of examples of the main findings.

4 RESULTS

We used the workflow for code-based clustering presented in the previous section to plot phylogenetic trees for the six top architectures in our dataset. We found that the current IoT malware scene is mainly invaded by three families tightly connected to each other: *Gafgyt*, *Mirai* and *Tsunami*. They contain hundreds of variants grouped under the same AV label and are the ones with longer persistence on VirusTotal. All three started to present fused traits over time and they still hit on VirusTotal. On the other hand, more specialized IoT malware targets specific CPU architectures and have a much shorter appearance. Today IoT malware code is not as complex as the one found in Windows malware, yet AVs may lose robustness when it comes to identifying widely reused functions and packed samples.

Table 3: Common functions across top10 malware families.

VS	Gafgyt	Mirai	Tsunami	Dnsamp	Hajime	Ddostf	Lightaidra	Pnscan	Skeeyah	VPNFilter
Gafgyt		115	189	3	1	2	18	-	-	-
Mirai			63	1	1	-	2	-	-	-
Tsunami				4	-	3	1	-	-	-
Dnsamp					-	65	-	-	-	-
Hajime						-	-	-	-	-
Ddostf							-	-	-	-
Lightaidra								-	-	-
Pnscan									-	-
Skeeyayh										-
VPNFilter										

As described in Section 3.5, the distance function we used for the HNSW algorithm is based on the number of functions with binary similarity ≥ 0.5 (as suggested by Diaphora). The analyst can then adjust this threshold when plotting the graphs to either display even uncertain similarities among families (at 0.5 threshold) or highlight only the perfect matches of exact code reuse (at 1.0 threshold).

4.1 Code Reuse

Figure 3 shows the lineage graph for MIPS samples plotted at similarity ≥ 0.9 and with node colored according to their AVClass labels.

Overall, MIPS samples include 39 different labels. However, the graph is dominated by few large families: *Gafgyt*, *Tsunami* and *Mirai*. These three families cover 87% of the MIPS samples and they are also the ones that served as inspiration for different groups of malware developers, most likely because of the fact their source code can be found online. It is interesting to note how this tangled dependency is reflected in the fact that the most of the *Tsunami* variants are located on the left side of the picture close to *Gafgyt*, but some of them appear also on the right side due to an increased number of routines borrowed from the Mirai code.

Besides these three main players, the graph also shows samples without any label or belonging to minor families. For example, the zoom region [A] contains a small connected component of 283 *Dnsamp* samples with a tail of 4 samples: 1 with label *Ganiw* and 3 with label *Kluh*. All together are linked to *ChinaZ*, a group known for developing DDoS ELF malware. The very high similarity between *Ganiw* and *Kluh* seems to be more interesting, since *Kluh* could be seen as an evolution of the first (and appeared 3 months after on VirusTotal), yet AVs assign them different labels.

Table 3 reports the number of shared functions (at 0.9 similarity) across the top 10 families in our dataset and takes into account

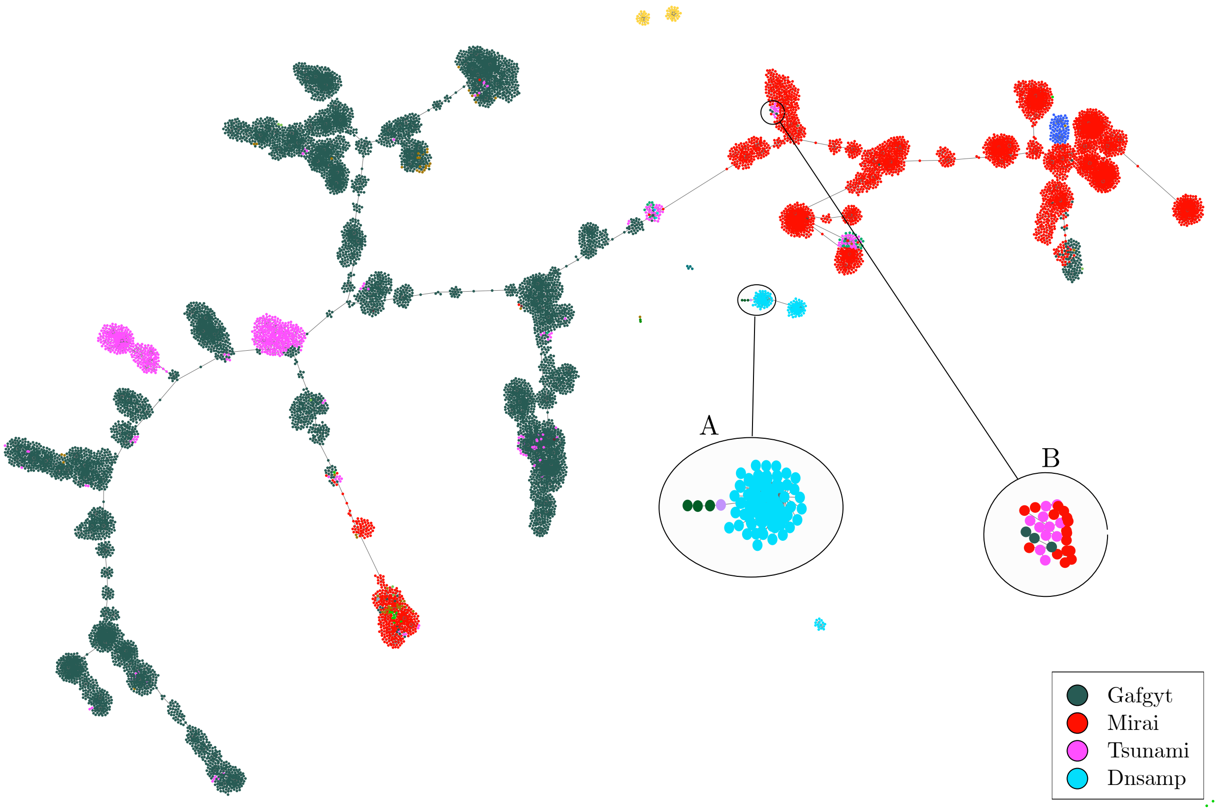

Figure 3: Lineage graph of MIPS samples colored by family.

Table 4: Outlier samples and AVClass labels

Architecture	**Number of samples**		
	Wrong label	**Without label**	**Total**
ARM 32-bit	19	9	28
MIPS I	25	41	66
PowerPC	1	4	5
SPARC	2	0	2
Hitachi SH	7	0	7
Motorola 6800	8	2	10
Total	62	56	118

the full picture of the six main architectures. The code sharing for *Mirai*, *Gafgyt* and *Tsunami* is once again confirmed to play a fundamental role in IoT malware with hundreds of functions shared across the three. However, we can see their incidence in minor families like *Dnsamp*, which borrows functions for random numbers generation and checksum computations, or *Lightaidra*, reusing 18 functions from *Gafgyt*. Less widespread families such as *Dnsamp* and *Ddostf* also show high similarity with a total of 65 shared functions. Instead, targeted campaigns like *VPNFilter* do not overlap with main components of the famous families.

4.2 Outliers and AV Errors

One of the analysis we can perform on the phylogenetic trees is the detection of *anomalous* labels, by looking for *outlier* nodes. We define as outlier a (set of) nodes of one color which is part of a cluster that contains only nodes of a different color. Outliers can correspond to samples that are misclassified by the majority of AV scanners or to variants of a given family that have a considerable amount of code in common with another family (and for which, therefore, it is difficult to decide which label is more appropriate). But outlier can also be used to assign a label, based on its neighbors, to samples for which AVClass did not return one.

Although the number of mislabelled samples is not significant in our dataset, we can use our automated pipeline to promptly detect suspicious cases in newly collected data. The outliers discussed in this section also show that a very high ratio of code similarity can often confuse several AV signatures.

Based on a manual inspection of each group of outliers, Table 4 reports a lower bound estimation of the mislabelling cases broken down by architecture. Overall we found 118 cases with 62 samples we believe to have a wrong AVClass label and 56 for which AVClass was not able to agree on the AV labels. ARM and MIPS (which cover 66% of our dataset) are responsible for over 80% of the errors, with MIPS samples being apparently the most problematic to classify. The pattern is reversed for less popular architectures, like Hitachi SH and Motorola 68000 (13.3% of the dataset) that account for

Table 5: Number of variants recognized for top 10 families in our dataset. Malware families with - contained *only* stripped samples which prevented any accurate variant identification.

Family	Candidate	Validated Variants (Source code)	Number of samples			Persistence (days)	
	Variants		Min.	Max.	Avg.	Max.	Avg.
Gafgyt	1428	140	1	4499	285.59	1210	283.21
Mirai	386	57	1	776	39.05	661	103.35
Tsunami	210	27	1	544	93.59	1261	421.63
Dnsamp	48	4	3	1394	362.75	1444	691.25
Hajime	1	1	1	1	1	1	1.00
Ddostf	11	3	2	755	260.00	483	308.33
Lightaidra	7	7	1	4	1.43	299	43.57
Pnscan	1	1	2	2	2.00	1	1.00
Skeeyah	-	-	-	-	-	-	-
VPNFilter	-	-	-	-	-	-	-
Total	240	2091					

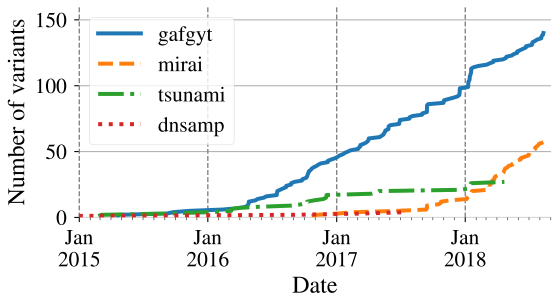

Figure 4: Appearance of new variants over time.

17 mislabelled samples, while PowerPC and SPARC (20.6% of the dataset) had only 7 cases.

Looking closer, all cases of wrong labels seemed to be due to a high portion of code reuse between two or more families. The zoom region [B] in Figure 3 is an example of this type of errors. A Tsunami variant that borrows a number of utility functions from Mirai resulted in few of its samples being misclassified as Gafgyt by many AV vendors.

Another example, this time related to a smaller family, is a set of 12 *Remaiten* samples that AVClass reported as *Gafgyt* (*Remaiten* is a botnet discovered by ESET that reuse both *Tsunami* and *Gafgyt* code, that extend with a set of new features). We also observed that in some cases AVs assign different labels for samples with an almost full code overlap. For example, under PowerPC, a binary is assigned the label *Pilkah*, thus giving birth to a new family, even if it is only a very minor variation of *Lightaidra*.

Finally, we found examples of how an extremely simple and well known packer like UPX can still cause troubles to AV software. For instance, 29 packed samples for MIPS did not get an AVClass label even if their code was very close to *Gafgyt*.

4.3 Variants

The phylogenetic trees produced by our method can also be used to identify fine-grained modifications and relationships among variants within the same malware family.

In order to bootstrap the identification of variants we decided to take advantage of the binary similarity-based symbol propagation described in Section 3.3. As a first step we identify candidate variants by grouping all malware samples based on their set of unique symbols. These symbols were either present in the binary (in case of an original unstripped binary) or were propagated from other unstripped binaries (in case of an original stripped binary). While this symbol-based variant identification technique is subject to errors – noise from symbol extraction, incomplete symbol propagation – it gives a first estimate of the number of variants by capturing fine-grained differences such as added, removed or renamed functions. Table 5 provides the number of identified variants for the top 10 largest malware families in our dataset. We can see that *Gafgyt*, and to a less extent *Mirai* and *Tsunami* appear to have spurred more than 2,000 variants all together. This phenomenon is supposedly fueled by the availability of the source code online for these three major malware families. It is important to note that given that this step relies on symbols it excludes all stripped samples for which symbols could not be propagated, e.g., all samples of the *VPNFilter* malware were stripped hindering the identification of variants.

As a second step we rely on the leaked source code collected from online repositories, as described in Section 3.4 to validate previously identified variants. By matching symbols found or propagated in the binaries with functions found in the source code we were able to validate more than 200 variants. It is interesting to see that as much as 50.3% of the samples had at least a partial match to our collected source code – but only 740 samples resulted in a perfect match of the entire code. This suggests that many malware authors take inspiration from leaked source code, yet they introduce new modifications, thus creating new independent variants. The surprisingly high number of variants having their source code online is a great opportunity for us to validate and better study them. Validating the others unfortunately require time-consuming manual analysis. From Table 5 we can see that the collected source code enabled us to validate 240 variants with *Gafgyt* taking a slice equal to 58% of the total, followed by *Mirai* with a lower share of almost 24%. The source code we collected matched also minor families. For example *Hajime*, known to come with stripped symbols, was found to have one sample referring to the *Gafgyt* and *Mirai* variant *Okane*, actually suggesting the *Hajime* sample was misclassified by AVs. In a similar way, two samples of *Pnscan* partially matched with a port scanner tool named like the family and available on GitHub. However, the authors of these samples introduced new functionalities to the original code. While the availability of IoT malware source code online facilitates the development of variants, it can also be leveraged to identify and validate them. Finally, in order to evaluate the accuracy of the source code matching we took an extra step and manually verified and confirmed some of the variants that matched source code.

Another important aspect to understand the genealogy of IoT malware is the combination of binary data with timing information. By measuring the first and last time associated to each variant we can get a temporal window in which the samples of each variants appeared in the wild (shown in the last two columns of Table 5). Here we can notice how quickly-evolving families like *Gafgyt* and *Mirai* tend to result in short-lived variants. For instance, *Gafgyt* variants appeared in VirusTotal for an average of 10 months, and *Mirai* variants for four. Instead, *Tsunami* and *Dnsamp* variants persisted for longer periods: respectively one year and two months the first and almost two years for the second. Figure 4 shows, in a cumulative graph, the number of new variants that appeared over time for the three main families. It is interesting to observe the almost constant new release of *Gafgyt* variations over time, the slower increase of *Tsunami* variants, and the rapid proliferation of *Mirai*-based malware in 2018.

5 CASE STUDIES

After showing our automated approach for systematic identification of code reuse in Section 3 and presenting an overview of the phylogenetic tree in Section 4, we now discuss in more details two case studies. We use these examples as an opportunity to provide a closer look at two individual families and discuss their evolution and the multitude of internal variants.

It is important to note that the exact time at which each sample was developed is particularly difficult to identify as malware could remain undetected for long periods of time. Since ELF files do not contain a timestamp of when they were compiled, we can only rely on public discussions and on the VirusTotal first submission time as source for our labeling. Some families are only discussed in blog posts by authors that did not submit their samples to VirusTotal. Previous research also found that for APT campaigns the initial VirusTotal collection time often pre-dates the time in which the samples are “discovered” and analyzed by human experts by months or even years [19]. Therefore, in our analysis we simply report the earliest date among the ones we found in online sources and among all samples submitted for the same variant to VirusTotal. However, this effort is only performed for presentation purpose, as we believe that detecting the similarities and changes among samples (the goal of our analysis) is more important than determining which ones came first.

Example I – Tsunami (medium-sized family)

Tsunami is a popular IRC botnet with DDoS capabilities whose samples represent almost 4% of our dataset. Its code is available online and gives birth to a continuous proliferation of new variants, sometimes with minimal differences, other times with major improvements (i.e., new exploits and new functionalities). Tsunami’s main goal is to compromise as many devices as possible to build large DDoS botnets. Therefore, we obtained samples compiled also for less common architectures such as Motorola 68K or SuperH. Overall, 76% of its samples are statically linked but with the original symbols in place. When constructing the genealogy graph of *Tsunami*, we not only took advantage of the extracted symbols from the binaries but we also cross-correlated them with available source code of multiple variants we scraped from online forums, as explained in Section 3.4. This way we were able to color the graph and assign a name to different variants.

The top part of Figure 5 shows the mini-graph for six different architectures. The main part of the figure further zooms in on the evolution of a group of 748 ARM 32-bit binaries. These samples all share the main functionality of *Tsunami* and therefore the functions for DDoSing and contacting the CnC remained the same across all of them.

On the most right of Figure 5, there is a visible section in which the vast majority of samples are labeled as *Kstd* according to the AV labels. With only two flooders, *Kstd* represents one of the oldest and most famous sub-family which acted as a skeleton and inspiration for newer malware strains. By moving left on the graph, we meet a fairly high dynamic area with binaries very similar to each other but with new features such as frequent updates and new flooders. The first samples in this group correspond to the *Capsaicin* sub-family, for which we performed a manual investigation to identify the new functionalities. *Capsaicin* includes 16 flooders based on TCP, UDP and amplification attacks. It uses *gcc* directly on the infected device, taking its presence for granted. Some *Tsunami* variants are also examples of inter-family code reuse, with code borrowed from both *Mirai* and *Gafgyt*. For example, *Capsaicin* borrows from Mirai the code for the random generation of IP addresses that is used to locate candidate victims to infect. Some *Tsunami* samples also perform horizontal movement reusing *Mirai*’s Telnet scanner or SSH scanners also found in *Gafgyt*, while others use open source code as inspiration (e.g., the *Uzi* scanner).

Moving left we then encounter the *Weebsquad* and *Uzi* variants. The first is a branch spreading over Telnet and SSH, for which we could not find any online source code that matched our samples. We named these variants based on the fact that they all included their name in the binaries. Interestingly some AVs on VirusTotal mislabeled these samples as *Gafgyt*, possibly because of the code-reuse between *Tsunami* and *Gafgyt* we mentioned earlier.

In the left side of Figure 5 we encounter *Kaiten*, another popular variant from which many malware writers forked their code to create their own projects. For instance, *Zbot* (bottom-left on the graph) is a *Kaiten* fork available on GitHub, in which the authors added two additional flooder components.

Our similarity analysis also recognizes *Amnesia*, a variant which was discovered by M. Malík of ESET in January 2017. This sub-family includes exploits for CCTV systems and it is one of the rare Linux malware adopting Virtual Machine (VM) detection techniques. Unlike most of the samples in the graph, *Amnesia* is stripped and dynamically linked. However, our system detected very high code similarity with another unstripped sample which uses the same CCTV scanner and persistence techniques, but without VM detection capabilities. Thanks to our symbol propagation technique we also managed to connect the *Amnesia* samples to the rest of the family graph.

Example II – Gafgyt (large family)

Gafgyt is the most active IoT botnet to date. It is comprised of hundreds of variants and is the biggest family in our dataset with 50% of the samples. It targets home routers and other classes of vulnerable devices, including gaming services [20].

We visualize the code-similarity analysis of samples for ARM 32-bit in Figure 6. Our system identified more than 100 individual

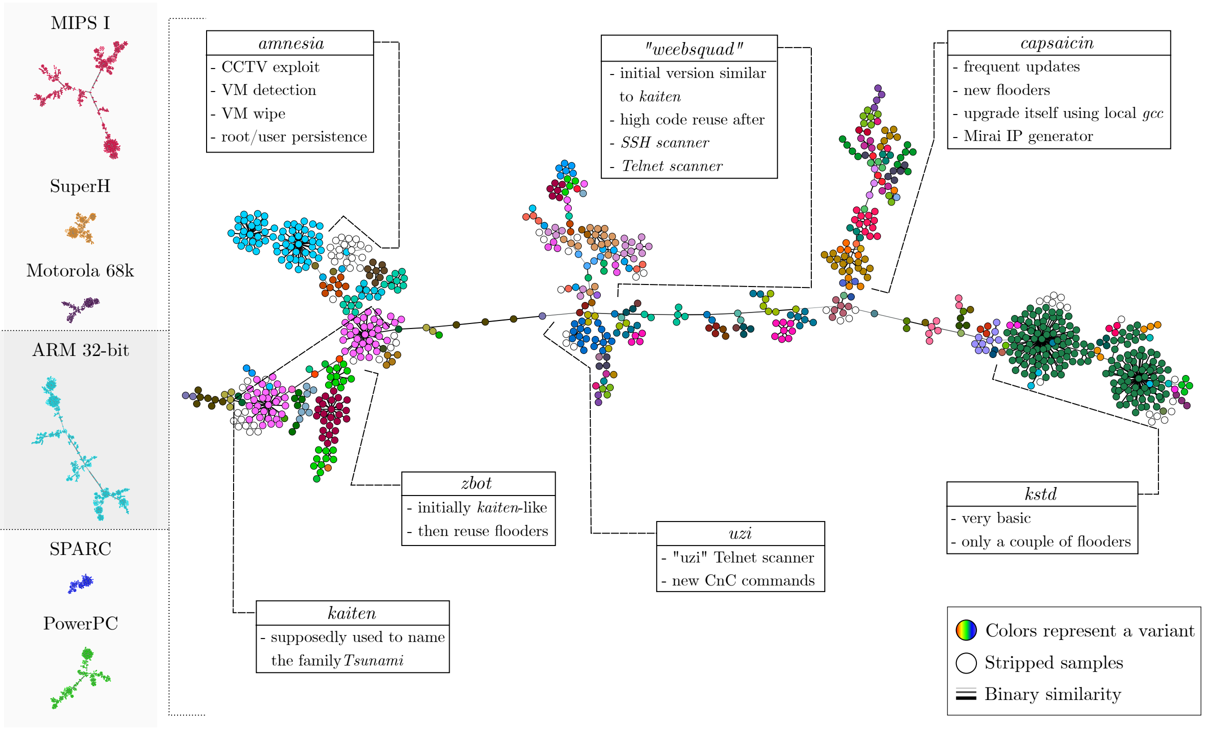

Figure 5: Lineage graph of *Tsunami* samples for *ARM 32-bit.*

variants. Like the Tsunami case study, we were often able to leverage available source code snippets to validate the identified variants.

The graph is clearly split into two main areas, with binaries compiled with *THUMB mode* support on the left, and with *ARM mode* only on the right. Since the two halves are specular we label variants separately on one or the other side of the graph to improve readability. *Bashlite* is believed to be one of the first variants of *Gafgyt*. Its samples are often mistaken for the *Qbot* variant (the two are frequently presented as a synonym) but their code presents significant differences. For example, *Qbot* uses two additional attack techniques (e.g., DDoS using the GNU utility `wget`). Our method rightfully recognizes them as belonging to the same family but as distinct variants.

The next cluster in our genealogy refers to *Razor*, which fully reuses the previous source code but adds an additional CnC command to clear log files, delete the shell history, and disable `iptables`. *Prometheus*, for which we crawled two bundles, is an example of malware versioning. Among the features of *Prometheus*, we see self upgrade capabilities and usage of *Python* scripts (served by the CnC) for scanning. Its maintainer added a Netis[3] scanner in *V4* to reinforce self propagation through exploitation. Self propagation and infection is further enhanced in *Galaxy* with a scanner dubbed *BCM* and one called *Phone* suggesting it targets real phones. Next to *Galaxy* we find an almost one-to-one fork we call *Remastered* which does a less intrusive cleanup procedure, cleaning temporary directories and history but without stopping `iptables` and `firewalld`.

Finally, in the top left-hand corner of Figure 6 we uncover *Angels*, reusing some of *Mirai*'s code for random IP generation (like other variants) and targeting specific subnets hard-coded in the binaries.

6 RELATED WORK

IoT Malware Landscape. Researchers have so far mostly focused on analyzing the current state of attacks and malware targeting IoT devices, usually by deploying honeypots [35, 45]. Antonakakis *et al.* [3] provided a comprehensive forensic study of the Mirai botnet, reconstructing it history and tracking its various evolutions. Cozzi *et al.* [12] carried out a systematic study of Linux malware describing some of the trends in their behavior and level of sophistication. This work included samples developed for different Linux-based OSes, without a particular focus on IoT systems. Different from this work, our objective is to study the relationships among IoT malware families (e.g., code reuse) and track sub-family variants and coding practices observed on a dataset an order of magnitude larger (93K samples vs. 10K) collected over a period of 3.5 years.

Malware lineage. First introduced in 1998 by Goldberg *et al.* [18], the concept of malware phylogenetic trees inspired by the study of the evolution of biological species. Karim *et al.*in [28] presented a code fragment permutation-based technique to reconstruct malware family evolution trees. In 2011, Dumitras *et al.* [15] presented

[3] Netis (a.k.a. Netcore in China) is a brand of routers found to contain an RCE vulnerability in 2014 [46].

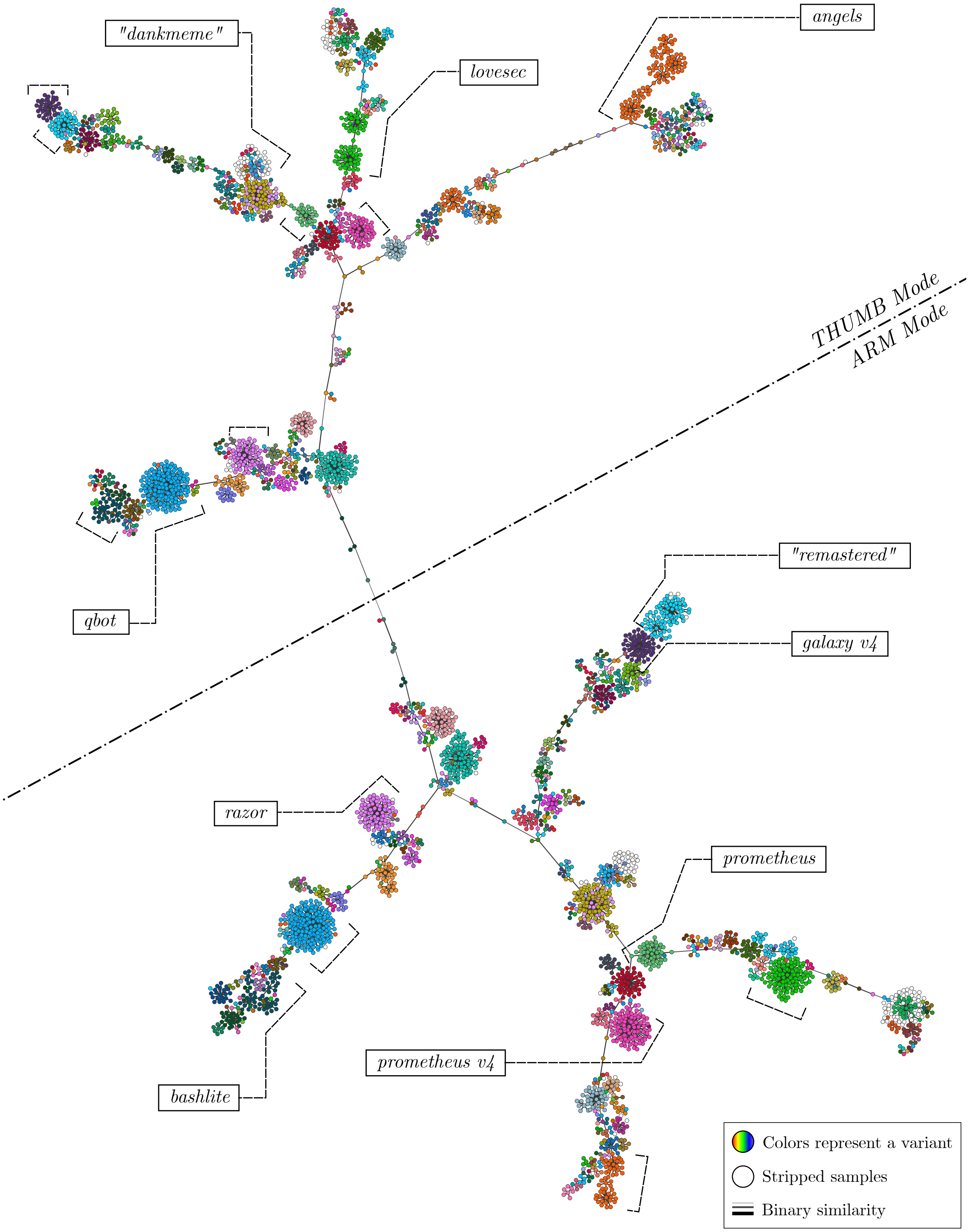

Figure 6: Lineage graph of *Gafgyt* samples for *ARM 32-bit*.

some guidance on malware lineage studies: (i) use of a combination of static and dynamic features, e.g., code fragments, dynamic CFGs, and (ii) use of time and source information related to studied samples. Lindorfer *et al.* [31] developed Beagle, a system designed to track the evolution of a malware family. They rely on dynamic analysis to extract the different functionalities – in terms of API calls – exhibited by a piece of malware. They then try to map these functionalities back to disassembled code so they can identify and characterise mutations in the malware family. Calleja *et al.* recently analyzed in [10] – extending from their previous work [9] – the evolution of 456 Windows malware samples observed over a period of 40+ years and identified code reuse between different families as well as with benign software. The types of code reuse they observed include essentially anti-analysis routines, shellcode, data such as credentials for brute-forcing attacks, and utility functions.

Recently, Haq *et al.* [21] reviewed 61 approaches from the literature on binary code similarity – some of which are used for malware lineage inference [24, 26, 33] – published over the last 20 years. While they purposely focus on academic contributions rather than binary diffing tools, they highlight the diversity, strengths and weaknesses of all these techniques. They also identify several open problems, some of which were faced in this work, such as scalability and lack of support of multiple CPU architectures. BinSequence [24] computes the similarity between functions extracted from binaries at the instruction, basic block and CFG levels. Authors applied their technique on different scenarios, including the identification of code reuse in two Windows malware families. They also claim a function matching accuracy higher than 90% and above state-of-the-art approaches such as Bindiff or Diaphora. iLine [26] is a graph-based lineage recovering tool based on a combination of low-level binary features, code-level basic blocks and binary execution traces. It is evaluated on a small dataset of 84 Windows malware and claim an accuracy of 72%. Ming *et al.* [33] also proposed an optimisation for the iBinHunt binary diffing tool, which computes similarity between binaries from their execution traces. They further apply their tool on a dataset of 145 Windows malware samples from 12 different families.

While these approaches for binary similarity and lineage inference provide invaluable insights when applied in the context in which they were developed, none of them can be applied on Linux-based IoT malware. First of all few of them are able to handle Linux binaries, and those that can typically do not go beyond the ARM and MIPS architectures. We also believe that binary-level or basic block-based malware slicing is likely to be prone to over-specific code reuse identification. Similarly, execution traces are likely to be too coarse-grained for variant identification. Additionally, we have witnessed in our dataset that, when used, packing of IoT malware can easily be evaded. As a result, given the reasonably low obfuscation of the IoT malware in our dataset we have decided to take this opportunity to use function-level binary diffing to identify relevant code similarities between and within IoT malware families. Finally, the lack of any available scalable Linux-compatible multi-architecture binary similarity technique led us to choose the open source binary diffing (IDA plugin) tool Diaphora [1].

7 CONCLUSION

We have presented the largest study known to date over a dataset consisting of 93K malicious samples. We use binary similarity-based techniques to uncover more than 1500 malware variants and validate more than 200 of them thanks to their source code leaked online. AV signatures appear to be not robust enough against small modifications inside binaries. As such rewriting a specific function or borrowing it from another family can be enough to derail AVs often leading to mislabeling or missed detections.

ACKNOWLEDGMENTS

We are greateful to Karl Hiramoto from VirusTotal for assisting us with the binary samples and VirusTotal reports used for this study. This research was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 771844 - BitCrumbs).

REFERENCES

[1] [n.d.]. Diaphora, a free and open source program diffing tool. http://diaphora.re/.
[2] [n.d.]. VirusTotal. https://www.virustotal.com/.
[3] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. 2017. Understanding the mirai botnet. In *USENIX Security*.
[4] T. Asano, B. Bhattacharya, M. Keil, and F. Yao. 1988. Clustering Algorithms Based on Minimum and Maximum Spanning Trees. In *Proceedings of the Fourth Annual Symposium on Computational Geometry* (Urbana-Champaign, Illinois, USA) *(SCG '88)*. Association for Computing Machinery, New York, NY, USA, 252–257. https://doi.org/10.1145/73393.73419
[5] Michael Bailey, Jon Oberheide, Jon Andersen, Z Morley Mao, Farnam Jahanian, and Jose Nazario. 2007. Automated classification and analysis of internet malware. In *RAID*.
[6] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. 2009. Scalable, behavior-based malware clustering.. In *NDSS*.
[7] BitDefender. 2018. New Hide 'N Seek IoT Botnet using custom-built Peer-to-Peer communication spotted in the wild. https://labs.bitdefender.com/2018/01/new-hide-n-seek-iot-botnet-using-custom-built-peer-to-peer-communication-spotted-in-the-wild/.
[8] BleepingComputer. 2019. Cr1ptT0r Ransomware Infects D-Link NAS Devices, Targets Embedded Systems. https://www.bleepingcomputer.com/news/security/cr1ptt0r-ransomware-infects-d-link-nas-devices-targets-embedded-systems/.
[9] Alejandro Calleja, Juan Tapiador, and Juan Caballero. 2016. A Look into 30 Years of Malware Development from a Software Metrics Perspective, Vol. 9854. 325–345.
[10] Alejandro Calleja, Juan Tapiador, and Juan Caballero. 2018. The MalSource Dataset: Quantifying Complexity and Code Reuse in Malware Development. (11 2018).
[11] Ricardo JGB Campello, Davoud Moulavi, and Jörg Sander. 2013. Density-based clustering based on hierarchical density estimates. In *PAKDD*.
[12] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. 2018. Understanding Linux Malware. In *IEEE S&P*.
[13] Matteo Dell'Amico. 2019. FISHDBC: Flexible, Incremental, Scalable, Hierarchical Density-Based Clustering for Arbitrary Data and Distance. arXiv:1910.07283 [cs.LG]
[14] Wei Dong, Charikar Moses, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*. ACM, 577–586.
[15] Tudor Dumitraş and Iulian Neamtiu. 2011. Experimental Challenges in Cyber Security: A Story of Provenance and Lineage for Malware. In *CEST*.
[16] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *KDD*.
[17] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proceedings of the VLDB Endowment* 12, 5 (2019), 461–474.
[18] Leslie Ann Goldberg, Paul W Goldberg, Cynthia A Phillips, and Gregory B Sorkin. 1998. Constructing Computer Virus Phylogenies. *J. Algorithms* 26, 1 (1998).
[19] Mariano Graziano, Davide Canali, Leyla Bilge, Andrea Lanzi, and Davide Balzarotti. 2015. Needles in a Haystack: Mining Information from Public Dynamic Analysis Sandboxes for Malware Intelligence. In *Proceedings of the 24rd USENIX Security Symposium (USENIX Security)*.

[20] M. Hao. [n.d.]. A Look into the Gafgyt Botnet Trends from the Communication Traffic Log. https://nsfocusglobal.com/look-gafgyt-botnet-trends-communication-traffic-log/.

[21] Irfan Ul Haq and Juan Caballero. 2019. A Survey of Binary Code Similarity. arXiv:1909.11424 [cs.CR]

[22] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1667–1680.

[23] Xin Hu, Kang G Shin, Sandeep Bhatkar, and Kent Griffin. 2013. Mutantx-s: Scalable malware clustering based on static features. In *USENIX ATC*.

[24] He Huang, Amr M. Youssef, and Mourad Debbabi. 2017. BinSequence: Fast, Accurate and Scalable Binary Code Reuse Detection. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS '17)*. ACM, 155–166.

[25] Jiyong Jang, David Brumley, and Shobha Venkataraman. 2011. BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *ACM CCS*.

[26] Jiyong Jang, Maverick Woo, and David Brumley. 2013. Towards Automatic Software Lineage Inference. In *22nd USENIX Security Symposium (USENIX Security 13)*. USENIX, Washington, D.C., 81–96. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/papers/jang

[27] Rommel Joven, Jasper Manuel, and David Maciejack. 2018. Mirai: Beyond the Aftermath. https://www.botconf.eu/wp-content/uploads/2018/12/2018-R-Joven-Mirai-Beyond-the-Aftermath.pdf.

[28] Md Enamul Karim, Andrew Walenstein, Arun Lakhotia, and Laxmi Parida. 2005. Malware phylogeny generation using permutations of code. *Journal in Computer Virology* 1 (11 2005).

[29] Dhilung Kirat and Giovanni Vigna. 2015. Malgene: Automatic extraction of malware analysis evasion signature. In *ACM CCS*.

[30] Peng Li, Limin Liu, Debin Gao, and Michael K Reiter. 2010. On challenges in evaluating malware clustering. In *RAID*.

[31] Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. 2012. Lines of Malicious Code: Insights into the Malicious Software Industry. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12)*. ACM, 349–358.

[32] Y. A. Malkov and D. A. Yashunin. 2018. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2018), 1–1. https://doi.org/10.1109/TPAMI.2018.2889473

[33] Jiang Ming, Dongpeng Xu, and Dinghao Wu. 2015. Memoized Semantics-Based Binary Diffing with Application to Malware Lineage Inference. In *IFIP Advances in Information and Communication Technology*, Vol. 455. 416–430. https://doi.org/10.1007/978-3-319-18467-8$_2$8

[34] A. Moser, C. Kruegel, and E. Kirda. 2007. Limits of Static Analysis for Malware Detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. 421–430.

[35] Yin Minn Pa Pa, Shogo Suzuki, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, and Christian Rossow. 2015. IoTPOT: analysing the rise of IoT compromises. In *WOOT*.

[36] PaloAlto Networks. 2019. Home & Small Office Wireless Routers Exploited to Attack Gaming Servers. https://unit42.paloaltonetworks.com/home-small-office-wireless-routers-exploited-to-attack-gaming-servers/.

[37] Leo Hyun Park, Jungbeen Yu, Hoo-Kong Kang, Taejin Lee, and Taekyoung Kwon. 2020. Birds of a Feature: Intrafamily Clustering for Version Identification of Packed Malware. *IEEE Systems Journal* (2020).

[38] Roberto Perdisci, Wenke Lee, and Nick Feamster. 2010. Behavioral Clustering of HTTP-based Malware and Signature Generation Using Malicious Network Traces. In *NSDI*.

[39] Roberto Perdisci and ManChon U. 2012. VAMO: Towards a Fully Automated Malware Clustering Validity Analysis. In *ACSAC*.

[40] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. *ACM Trans. Database Syst.* 42, 3, Article 19 (July 2017), 21 pages. https://doi.org/10.1145/3068335

[41] Marcos Sebastian, Richard Rivera, Platon Kotzias, and Juan Caballero. 2016. AVclass: A Tool for Massive Malware Labeling. In *RAID*.

[42] Symantec. 2018. Symantec Internet Security Threat Report (ISTR). https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-2018-en.pdf.

[43] Symantec. 2019. Symantec Internet Security Threat Report (ISTR). https://www.symantec.com/content/dam/symantec/docs/reports/istr-24-2019-en.pdf.

[44] Talos. 2018. New VPNFilter malware targets at least 500K networking devices worldwide. https://blog.talosintelligence.com/2018/05/VPNFilter.html.

[45] Pierre-Antoine Vervier and Yun Shen. 2018. Before Toasters Rise Up: A View into the Emerging IoT Threat Landscape. In *RAID*.

[46] T. Yeh. [n.d.]. Netis Routers Leave Wide Open Backdoor. https://blog.trendmicro.com/trendlabs-security-intelligence/netis-routers-leave-wide-open-backdoor/.

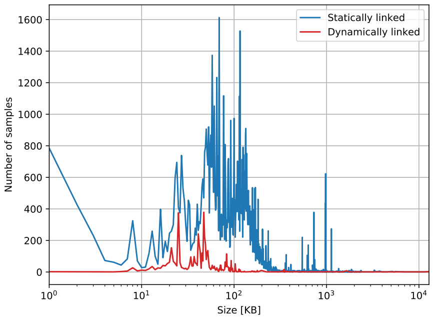

Figure 7: File size distribution of malware in the dataset.

A FEATURES-BASED CLUSTERING

In this Section we describe our initial attempt at reconstructing IoT malware lineage using a traditional feature-based clustering approach. As explained in Section 3, we eventually adopted a different solution to reach our goal. However, as feature-based clustering is often used in malware studies, we believe there is a value in reporting the results of this attempt and discuss the reasons behind its failure.

A.1 Foreword on Malware Clustering

Malware clustering has been extensively studied in order to cope with the increasing sophistication and the rapid increase in the number of observed samples [5, 6, 23, 25, 29, 38]. As a result, there's a long list of works (of which we summarize what we believe to be the most relevant ones).

A large corpus of works focus on behavior-based malware clustering [5, 6, 29, 38] and typically differ by their used malware features, clustering algorithm and size of the dataset. Bailey *et al.* [5] created fingerprints from user-visible system state changes (e.g., files written, processes created) and then leveraged a single-linkage hierarchical clustering algorithm to automatically classify approximately 3.7K samples. Bayer *et al.* [6] leveraged augmented malware execution traces and then applied a single-linkage hierarchical clustering algorithm on 14K samples. Perdisci *et al.* [38] produced malware network signatures by using clustering to extract structural similarities in malicious HTTP traffic traces generated by 25K samples. Kirat *et al.* [29] built a system that automatically generates system call-based signatures for 3.1K evasive malware samples and further grouped those samples using a complete-linkage clustering algorithm.

Others have looked at static analysis-based malware clustering [23, 25]. Hu *et al.* [23] proposed MutantX-S to exploit a hashing trick to reduce static feature dimension and leverage a prototype-based clustering algorithm to resolve the scalability issues faced by previous malware clustering approaches. Similarly, Jiang *et al.* [25] proposed BitShred to use feature hashing to reduce the high-dimensional feature spaces that are common in malware analysis.

Finally, Li *et al.* [30] discussed the challenges in evaluating malware clustering, especially when it comes to building an accurate ground truth. Perdisci *et al.* [39] also proposed a machine learning-based system to build an AV label-based model against which third party clustering results can be evaluated.

Note that none of these works have applied their technique on Linux malware, which brings a lot of challenges related to the various CPU architectures. Moreover, the size of our dataset (93K samples) and the high number of (static and dynamic) features led us to choose the FISHDBC [13] algorithm for our initial feature-based clustering.

A.2 Feature Extraction

To analyze each sample, we leverage a free ELF binary analysis service[4] based on a recent work [12]. The service relies on a combination of static and dynamic analyses to comprehensively evaluate ELF binaries. It provides runtime behavioral reports via its multi-architecture sandboxing environment, from which we extract 146 features that belong to five groups. We refer the reader to Appendix B for the complete list of extracted features.

1. **ELF and byte-level** features capture low-level characteristics of the binary, such as its architecture, whether it is statically or dynamically linked, stripped or unstripped, the number of ELF sections, its file size, the entropy of each section and its most common bytes, etc.
2. **Binary disassembly** features report numerical statistics extracted with IDA Pro, such as the number of functions, their complexity, the number of instructions, etc.
3. **Strings** includes printable strings extracted from the binary, grouped into IP addresses, URLs, and UNIX paths.
4. **Runtime behavior** covers the information extracted from the execution of the binary in a sandbox, including whether the sample was executed correctly, the list of issued system calls, the different files opened, modified or deleted, whether the binary has attempted to achieve persistence on the system, etc.
5. **Network traffic** features provide a detailed breakdown of all network connections observed while the binary was running, as extracted by the Zeek (formally Bro) IDS, including contacted IP addresses, files transferred, domain name resolved, etc.

A.3 Clustering

Our dataset is large and very complex, containing 93K samples and 146 features, several of them categorical. We converted categorical features to numeric ones with the standard *one-hot encoding* technique, whereby each categorical feature becomes a set of *n* boolean representing whether each item belongs to each of the *n* categories for that feature. For categorical features, we ended up with a sparse matrix having tens of thousands of columns: such a large dimensionality is generally very problematic in terms of scalability for generic clustering algorithms. To deal with it, we use FISHDBC [13], a density-based clustering algorithm designed for scalability for complex datasets and arbitrary/non-metric distance functions. FISHDBC approximates HDBSCAN* [11], an evolution of the widely known DBSCAN algorithm [16, 40], without generally compromising in terms of results quality. Due to scalability issues we could not run HDBSCAN* on our complete dataset, but we confirmed that results of FISHDBC and HDSBCAN* were equivalent on smaller datasets. This algorithm outputs hierarchical clustering results in a top-down approach—from the most coarse-grained to the most fine-grained—and allows to identify the level that yields the best classification.

We consider numeric and categorical features for each group separately; for categorical features we pre-process the dataset using *tf-idf* and the *Cosine distance*, while we use the *Euclidean distance* for numerical features. To empirically assess the impact of feature groups, we performed 25 rounds of clustering including different combinations of feature groups, i.e., by including or discarding some of the five categories.

To get a rough estimation of the quality of the clustering we use AV labels as a provisional ground truth. In fact, even if some errors in the label may exist, we still expect to find samples in the same cluster to largely come from the same family. By using the output of AVClass, we flag each cluster as one of four categories: (i) *Pure* if it contains all samples with the same AV label, (ii) *Single* if it contains a combination of samples with the same AV label and unlabelled samples, (iii) *Majority* if more than 90% of samples in the cluster have the same AV label, and (iv) *Mixed* if it does not fit any of the previous categories. Table 6 provides a summary of the results of the 25 rounds of clustering. For the sake of conciseness, we only provide the best results obtained per combination of feature groups across all tested weights. Note that the clustering on the IDA Pro features could only be performed on a restricted set of the 4,960 samples dynamically linked samples, to avoid introducing noise in the IDA Pro features due to the large amount of embedded library code. Moreover, the table does not contain results for the network features alone because network features were too sparse and could not be used by themselves to build our hierarchical clusters.

Table 6 shows that individual sets successfully identify several groups of samples belonging to the same family (i.e., *pure* clusters), but then also cluster together many samples that have little or nothing in common (e.g., *mixed* clusters). The results do not improve much by combining all features, as the limitation of each group tends to increase the noise in the overall classification. Out of all combinations we tried in our experiments, the ELF and bytes features alone produced the best clustering results with a total of 44,491 samples in *pure* clusters and only 14,204 samples in *mixed* clusters. However, even in this case roughly one third of our dataset was placed in *majority* clusters which erroneously contained samples of different families.

We then performed an investigation on the resulting clusters produced by the different feature group combinations. Here we wanted to understand whether these clusters could be directly used to group together samples that belong to the same variant or subfamily and, if the answer is affirmative, what exactly was changed between one version and the other. We first looked at the *pure clusters*. We noticed that *all* medium-to-large size malware families were broken down by our system in many *pure clusters*. If we consider the combination that produced the best clustering results, i.e., the ELF and bytes combination, 20,027 *Gafgyt* samples were clustered in 1,071 different pure clusters. Also, as many as 13,391 *Mirai*

[4] Padawan: https://padawan.s3.eurecom.fr

Table 6: Clustering results: static and dynamic features.

Feature groups					Clusters (# samples)			
ELF	IDA Pro	strings	behaviour	network	*pure*	*single*	*majority*	*mixed*
✓					**44,491**	**4,657**	31,649	14,204
	✓				3,677	45	316	1,082
		✓			18,141	3,120	23,412	50,328
			✓		27,889	1,097	5,726	60,289
✓	✓	✓	✓	✓	34,313	2,337	12,741	45,610
✓	✓	✓	✓		38,825	3,062	24,234	27,531
✓	✓	✓			39,904	2,495	17,667	33,586
✓	✓				42,427	2,587	**34,118**	14,520
		✓	✓	✓	20,822	983	12,964	58,883

samples populated a total of 654 pure clusters. Initially, this would make them good candidates for our sub-family investigation. As expected, indeed different clusters often captured different common features of the samples. For example, they separated dynamically vs statically linked binaries, or those samples that successfully executed in our VM from those that did not (and therefore resulted in an empty dynamic behavior profile). However, our goal was not to distinguish *Mirai* samples that were dynamically or statically linked, but rather identify its evolution over time. Unfortunately, the resulting clusters did not capture our need to isolate sub-families but rather samples that produced similar features (e.g., two samples that immediately terminate with an error message are not necessarily similar, despite the common behavior). During the manual investigation of the clustering results, we also noticed that the captured runtime and network behavior of different variants of the same family, when not missing, were often identical or so similar that the clustering algorithm would hardly differentiate them. For example, most variants of Mirai would follow the same high-level process after the device is compromised: (i) reach out to the C&C server, (ii) retrieve some target IP addresses to scan for worm-like replication, (iii) launch scanning, (iv) receive DDoS attack target(s), and (v) launch DDoS attack(s). This hinders the identification of variants from such a trace. Additionally, considering finer-grained features is likely to introduce overly specific clusters.

We also manually investigated those clusters that contained samples with different AV labels. In particular, we looked at those that had a predominant number of samples with a consistent AV label, and a small number of samples with a different one (*majority clusters*), e.g., `(gafgyt: 33), (aidra:2)`. While intuitively this could have been the result of errors in AV classifications, after dozens of manual investigations we could not find a single mislabeled sample. Please remember that this does not mean there were no errors in individual AV labels (we did find several of those), but that by applying the majority voting provided by AVclass the result (when a consensus was reached) was always correct. Errors in the majority voting also existed, as explained in more details in Section 4.2, but we needed a more precise clustering to successfully isolate them from the noise.

Traditional clustering based on static and dynamic features was insufficient to identify meaningful similarities and isolate variations among sub-families. In particular, when applied to a large dataset, the number of errors largely exceeded the ability to manually investigate and correct the results. Dynamic features (for example those extracted from runtime behaviour or network traffic) failed to accurately classify samples even into coarse-grained malware families. On the other hand, we observed that static features (for instance ELF features) would produce very compact micro clusters sensitive to very fine-grained changes in the binary representation of malware samples. While this was more successful to group together samples belonging to the same family, such over-sensitive classification turned out to be inappropriate for the identification of IoT malware variants. This contrasts with previous clustering and lineage works e.g., on Windows [37], where malware programs express more unique behaviors compared to the IoT counterpart seen to date.

B LIST OF STATIC AND DYNAMIC FEATURES

Feature name: **Description**

bytes.common_bytes: List of the three most common bytes (with counter)
bytes.entropy: The entropy of the binary
bytes.header: First 16 bytes of the file
bytes.footer: Last 16 bytes of the file
bytes.longest_sequence.length: Longest sequence of the same byte (byte, offset, length)
bytes.min_entropy: Lowest entropy among 16K bytes blocks
bytes.max_entropy: Highest entropy among 16K bytes blocks
bytes.null_bytes: Number of null (0) bytes
bytes.printable: Number of printable bytes
bytes.rarest_bytes: List of the three rarest bytes (with counter)
bytes.unique_bytes: Number of unique bytes (0-255)
bytes.white_spaces: Number of white-spaces (0x32,\n,\r,\t) bytes

elf.anomalies.ehph_diff: Difference between segment virtual address and file offset
elf.anomalies.entrypoint.permission: Anomalous entrypoint: Permission
elf.anomalies.entrypoint.section: Anomalous entrypoint: Section
elf.anomalies.entrypoint.segment: Anomalous entrypoint: Segment
elf.anomalies.sections.cpp_prelink: Anomalous sections: C++ prelink section
elf.anomalies.sections.grub_module: Anomalous sections: Grub module
elf.anomalies.sections.headers: Anomalous sections: Wrong number of section headers
elf.anomalies.sections.high_entropy: Anomalous sections: High entropy
elf.anomalies.sections.kernel_object: Anomalous sections: Kernel object
elf.anomalies.sections.section_header_null: Anomalous sections: Null section headers
elf.anomalies.sections.shentsize_empty: Size of section header table's entry null
elf.anomalies.sections.shnum_empty: Anomalous sections: Number of section headers empty
elf.anomalies.sections.shnum_pastfile: Anomalous sections: Section header table beyond file
elf.anomalies.sections.shoff_empty: Anomalous sections: Section header table offset empty
elf.anomalies.sections.shoff_pastfile: Anom. sec.: Section header table offset beyond file
elf.anomalies.sections.uncommon: Anomalous sections: Uncommon sections
elf.anomalies.sections.wrong_shstrndx: Anom. sec.: Wrong section name string table index
elf.anomalies.segments.error: Error in segments table
elf.anomalies.segments.headers: Anomalous segments: Wrong number of program headers
elf.anomalies.segments.high_entropy: Anomalous segments: High entropy
elf.anomalies.segments.high_mem: Segment memory size much bigger than physical size
elf.anomalies.segments.wx: Anomalous segments: `W&X` permission
elf.class: ELF file's class
elf.comment: .comment section of the ELF, if present
elf.data: Data encoding of the-specific data
elf.debug: If the binary contains debug information (compiled with -g)
elf.dynfuncs: Dynamic symbols being used, of type FUNC in particular
elf.entrypoint: Binary entrypoint
elf.e_phentsize: Size in bytes of one entry in the program header table
elf.e_phnum: Number of entries in the program header table
elf.e_phoff: Program header table's file offset in bytes
elf.e_shentsize: Size in bytes of one entry in the section header table
elf.e_shnum: Number of entries in the section header table
elf.e_shoff: Section header table's file offset in bytes
elf.e_shstrndx: Index of section header table containing section names
elf.gdb: Error raised by gdb
elf.interpreter: ELF's declared interpreter
elf.link: Statically or dynamically linked
elf.machine: Required architecture for the file
elf.malformed.entrypoint: Malformed ELF: Wrong entrypoint
elf.malformed.pastload: Malformed ELF: Beyond LOAD segment
elf.malformed.pastphnum: Malformed ELF: Beyond program header table
elf.malformed.pastsegment: Malformed ELF: Beyond segment
elf.needed: DT_NEEDED entries for dynamic ELF files
elf.note: .note.* sections of the ELF, if present
elf.nsections: Number of sections
elf.nsegments: Number of segments
elf.osabi: Operating system/ABI identification

elf.pyelftools: Exception raised by pyelftools, if any

elf.readelf: Error raised by readelf

elf.soname: PT_SONAME entry for dynamic ELF files

elf.stripped: Whether the binary has been stripped or not

elf.stripped_sections: Whether the sections table of the binary has been stripped or not

elf.type: Object file type

---

strings.ip: Potential IPs (v4 and v6) found in the binary

strings.path: Potential UNIX paths found in the binary

strings.url: Potential URLs found in the binary

---

idapro.average_bytes_func: Average size in bytes of a function

idapro.avg_basic_blocks: Average number of basic blocks respect to functions

idapro.avg_cyclomatic_complexity: Average cyclomatic complexity respect to functions

idapro.avg_loc: Average lines of code respect to functions

idapro.branch_instr: Number of branch instructions

idapro.bytes_func: Total size in bytes of the functions

idapro.call_instr: Number of call instructions

idapro.func_loc: Percentage of instructions belonging to functions

idapro.indirect_branch_instr: Number of indirect branch instructions

idapro.loc: Explored lines of code

idapro.max_basic_blocks: Max basic blocks

idapro.max_cyclomatic_complexity: Max cyclomatic complexity

idapro.nfuncs: Number of functions detected

idapro.percent_load_covered: Percentage of covered load segment

idapro.percent_text_covered: Percentage of covered text section

idapro.syscall_instr: Number of syscall instructions

---

behavior.user.argv0_rename: Procs renaming argv0

behavior.user.askroot: Wheter the execution got permission related errors

behavior.user.checkgid: If gid is checked

behavior.user.checkuid: If uid is checked

behavior.user.cmds: System cmds

behavior.user.compare: `strcmp` or `memcmp` comparison

behavior.user.cve: Possible CVEs exploited

behavior.user.dropped.create: Dropped files: Create

behavior.user.dropped.link: Dropped files: Link

behavior.user.dropped.linkfrom: Dropped files: Link from

behavior.user.dropped.modify: Dropped files: Modify

behavior.user.empty: Empty or no trace

behavior.user.errors.enosys: Errors from execution: Syscall not implemented

behavior.user.errors.execfault: Errors from execution: Execution fault

behavior.user.errors.illegal: Errors from execution: Illegal instruction

behavior.user.errors.missinglibs: Errors from execution: Missing library

behavior.user.errors.segfault: Errors from execution: Segmentation fault

behavior.user.errors.sigbus: Errors from execution: Bus error

behavior.user.errors.wronginterp: Errors from execution: Wrong interpreter

behavior.user.ioctl.fail: Ioctls: Fail

behavior.user.ioctl.success: Ioctls: Success

behavior.user.ioctl.total_no: Ioctls: Total number

behavior.user.libccalls.total_no: Libc calls from execution: Total number

behavior.user.libccalls.unique: Libc calls from execution: Unique

behavior.user.libccalls.unique_no: Libc calls from execution: Unique number

behavior.user.lineslost: Amount of trace lines not correctly parsed

behavior.user.persistence.create: Sample persistence: Create

behavior.user.persistence.link: Sample persistence: Link

behavior.user.persistence.linkfrom: Sample persistence: Link from

behavior.user.persistence.modify: Sample persistence: Modify

behavior.user.proc_rename: Procs renaming

behavior.user.procs: Number of processes spawned

behavior.user.ptrace_request: Ptrace requests

behavior.user.read_only: Files being read

behavior.user.rooterr.EACCES: EACCES type of permission related error

behavior.user.rooterr.EPERM: EPERM type of permission related error

behavior.user.sleep_max: Max sleep

behavior.user.syscalls.total_no: Syscalls from execution: Total number

behavior.user.syscalls.unique: Syscalls from execution: Unique

behavior.user.syscalls.unique_no: Syscalls from execution: Unique number

behavior.user.unlink: Unlink files

behavior.user.unlink_itself: Unlink itself

---

dynamic.error: Errors encountered during sandboxing

dynamic.stderr: Standard output during analysis

dynamic.stdout: Standard error during analysis

---

nettraffic.conn.avg_duration: Average duration of connections

nettraffic.conn.bytes: Number of bytes exchanged

nettraffic.conn.conns: Number of connections

nettraffic.conn.ips: List of unique IP addresses contacted

nettraffic.conn.pkts: Number of packets exchanged

nettraffic.conn.ports: List of unique destination ports

nettraffic.dns.qry_resp: List of unique DNS queries and their responses

nettraffic.dns.queried_domains: List of unique domains resolved through DNS

nettraffic.files.dropped_files_hash: List of unique hashes (SHA-256) of dropped files

nettraffic.files.dropped_files_mimetype: List of unique MIME types of dropped files

nettraffic.files.dropped_files_source_ips: List of unique IP addresses from which dropped files have been downloaded

nettraffic.files.dropping_protos: List of unique protocols used to drop files

nettraffic.ssl.ssl_domains: List of unique domains contacted over SSL/TLS
