Devils in the Clouds: An Evolutionary Study of Telnet Bot Loaders

Yuhui Zhu*†, Zhenxiang Chen *†‖, Qiben Yan‡, Shanshan Wang*†, Alberto Giaretta§,
Enlong Li*†, Lizhi Peng *†, Chuan Zhao *†, Mauro Conti¶
*Shandong Provincial Key Laboratory of Network Based Intelligent Computing, University of Jinan, China
†School of Information Science and Engineering, University of Jinan, China
‡Department of Computer Science and Engineering, Michigan State University, USA
§AASS MPI Lab, Örebro University, Sweden
¶Department of Mathematics, University of Padua, Italy
‖Corresponding author, Email: czx.ujn@gmail.com

***Abstract*—One of the innovations brought by Mirai and its derived malware is the adoption of self-contained loaders for infecting IoT devices and recruiting them in botnets. Functionally decoupled from other botnet components and not embedded in the payload, loaders cannot be analysed using conventional approaches that rely on honeypots for capturing samples. Different approaches are necessary for studying the loaders evolution and defining a genealogy. To address the insufficient knowledge about loaders' lineage in existing studies, in this paper, we propose a semantic-aware method to measure, categorize, and compare different loader servers, with the goal of highlighting their evolution, independent from the payload evolution. Leveraging behavior-based metrics, we cluster the discovered loaders and define eight families to determine the genealogy and draw a homology map. Our study shows that the source code of Mirai is evolving and spawning new botnets with new capabilities, both on the client side and the server side. In turn, shedding light on the infection loaders can help the cybersecurity community to improve detection and prevention tools.**

***Index Terms*—IoT botnet, loader, taxonomy, lineage inference**

I. Introduction

Following the growth of the IoT market, botnets recruiting IoT devices have become a major cyber-security threat. Earlier in 2016, the emerging Mirai botnet drew attention from the cybersecurity community. The operator launched a 1.1Tbps DDoS attack using 148,000 IoT devices, breaking the record and making it the most notorious botnet clan in the following years. As the Mirai's code release [1] have stimulated the evolution of botnet malware, the cybersecurity community has invested considerable energy to define complete taxonomies that would help to understand the differences and similarities between emerging variants. Empirical studies have discovered multiple *families* and defined by general taxonomy studies [2]–[4], while other efforts went into collecting and dissecting malware samples to identify less obvious *variants* [5]–[7].

Although these investigations depicted a clear genealogy of malware families and variants, sample-centered approaches fail to consider server-side components that are characteristics of Mirai-like botnets. In particular, while conventional worms infect new victims independently, Mirai exhibits a decoupled design that assigns infection functions to a separated self-contained *loader server*, deployed on cloud services. By assigning to the bots the discovery task and offloading the infection process to an external loader server, Mirai reduces the amount of resources required for a machine to function as a bot and allows for recruiting resource-restricted IoT devices.

The design choice of Mirai presents different challenges for the research community. On the one hand, delegating the infection tasks to cloud services results in botnets that are split in disjoint parts, making them harder to study as a whole system. On the other hand, this choice prevents honeypots from capturing a vital part of Mirai code and operations. Therefore, studies limited to payload-based lineage inference are inherently incomplete, as they neglect the infecting toolkits. Server-side studies are critical to understand in depth botnets infrastructure. By highlighting the peculiar characteristics of intrusion toolkits, malware studies can shed new light on the relationship between botnet campaigns, as well as improve the efficiency of defense strategies against new attack vectors.

In this paper, we focus on the telnet loader, the only infection toolkit distributed in the original Mirai codebase. Our work provides a server-side view of botnets evolution and a novel behavior-based taxonomy of bot loaders, following a conventional family-variant epistemology. To address the absence of loader samples, we analyse the interaction logs captured using telnet honeypots. Under the assumption that different intrusion toolkits use different infection instructions, we adopt a semantic-aware strategy to map differences and similarities in instruction sets to lineages. The paper's contributions are the following:

- We propose a semantic-aware method to analyse the lineage of bot loaders through their interaction logs, captured via honeypots;
- We conduct a taxonomy study on infection toolkits, evaluate the behavioural patterns, and define a genealogy of eight families;
- We highlight the existence of an unconventional loader, suspected to conduct fileless attacks, and we confirm its homology with conventional file-based bot loaders;
- We highlight that infection loaders evolve independently

from their payloads and we advocate the importance of a server-side perspective in botnet provenance attribution.

II. Related Work

Since the source code of Mirai has been publicly released, dozens of variants appeared in the wild and hundreds of massive botnets spawned. The cybersecurity community strove to produce a taxonomic view on these botnet campaigns and capture their evolution. Most studies built their observations on the relationships between botnets on empirical definitions of *families* and *variants*. Pa et al. [8] analyzed and categorized emerging botnets based on an observation of shared command sequence patterns. Antonakakis et al. [2] and Herwig et al. [3] analyzed two emerging botnet families, Mirai and Hajime, to discuss their propagation and evolution. To examine the competition and battle among botnets, Griffioen [4] categorized botnet campaigns into several variants by their identity strings before discussing their behavior. Dang et al. [9] categorized fileless attacks on Linux-based IoT devices and correlated these attack vectors with known botnet families. Alrawi et al. [10] discussed the lifecycles of botnets based on family definitions from VirusTotal [11], a popular malware detection tool with a collection of anti-virus engines.

Beyond the definition of families, depicting their evolution and variation under the family-variant epistemology is also critical to botnet studies. Most studies obtain evidence from bot samples, the most easy-to-access components, by involving *bindiff* and other sample-centered techniques. Wang et al. [7] pointed out that investigating relationships among botnet families could be a fundamental step for provenance, triage, labeling, lineage analysis, and authorship attribution. They derived knowledge of botnet samples from online articles and captured samples, proposed a hybrid methodology to construct a lineage graph, and discussed the lineage of 72 botnet families. Cozzi et al. [5] shed light on the tangled genealogy of botnet samples by measuring shared components across malware samples from different families. Most anti-virus engines also used YARA [12] to match the shared patterns of a malware family, so that they could relate unseen samples to existing families or variants.

As dissecting samples of bot loaders is impractical due to the absence of samples, many studies further explored various intrusion fingerprints to reveal their covert relationships. The first studies on Mirai [2], and Hajime [3] studied the password dictionaries captured by honeypots to discuss the lineage of botnet variants. Lingenfelter et al. [13] made a comparison of initial commands and query tokens to demonstrate the variation of telnet intrusion toolkits. Torabi et al. [14] tried mining unique strings from logs to build associations among active botnets. Tabari et al. [15] made a statistical analysis of the most commonly exploited vulnerabilities, credentials, and intrusion commands. However, while sample-centered works have always overlooked loaders in lineage inference, simply comparing strings or fingerprints in intrusion toolkits cannot yield a systematic view. The families and variants derived from malware samples also brought a prior hypothesis bias to these studies, which effectively hinders the understanding of the desired server-side behaviors.

III. Methodology

In this section, we categorize telnet loaders into families and discuss their variation through captured sequences of intrusion commands. We assume that the sequence of intrusion commands may reflect loaders' functions and inner implementation, so we propose a semantic-aware metric to describe the similarity among collected sequences and leverage agglomerative clustering to categorize them into families. Based on the agglomerative tree, we further present the shared patterns among sibling loaders to yield a systematic conclusion about the variation and homology of intrusion toolkits from a server-side perspective.

A. Data Collection

Telnet is a text-based protocol commonly used for accessing a remote shell on IoT platforms. Although botnets have been evolving their toolkits to exploit new vulnerabilities in different protocols, bot masters are still working on telnet-based intrusions to adapt to more vulnerable devices. Based on such behaviors, recent studies on IoT botnets [3], [4], [14] all considered telnet loaders as a crucial basis to make comparisons among botnet families. Thus, we initialize the study by investigating telnet protocols to understand the behavior of botnet loaders.

We deploy a honeycloud system to record command sequences from loaders. We deploy frontends on 3 virtiual cloud servers in China, Singapore, and the United States to redirect requests to the honeycloud backend. The honeycloud backend dispatches telnet conversations to the backing devices listed in Table I, then it substitutes the requested username and password to allow botnets to access our deployed devices. We only record the requests of intrusion commands from botnets and drop all responses to avoid client-side noise for our server-side analysis. All requests collected from a conversation are concatenated into a single "request log" to represent the behavior and function of a loader.

TABLE I
Deployed backing devices during the experiment

Type	Device name	Software version
Smart router	Lenovo Y1S	PandoraBox git-6fcbaa5
	Netgear R7800	OpenWRT 21.02
	Netgear R6300v2	KoolShare Merlin
IP Camera	Hikvision	(Stock)
ONU	CMCC I-120EM	(Stock)
Other	Raspberry Pi 3B	Raspberry Pi OS Lite Jan 2021

To evaluate the effectiveness of our proposed method, we run a Hajime bot and a Mirai loader in a QEMU ARM sandbox to generate control group data. The Hajime bot sample is provided by MalwareBazaar[1].

[1]https://bazaar.abuse.ch/

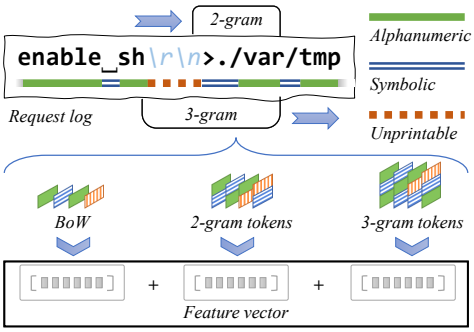

Fig. 1. Byte-based tokenization and n-gram vectorization of request logs.

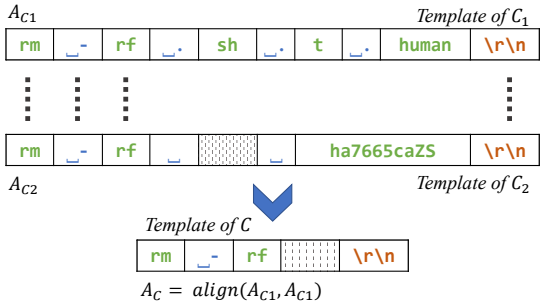

Fig. 2. Smith-Waterman algorithm on the agglomerative tree.

B. Feature Extraction and Dissimilarity Measurement

In this step, We adopt classical methods from Natural Language Processing (NLP) to embed request logs into a feature space, then quantify the dissimilarity between any two of them. Fig. 1 illustrates the process of feature extraction.

Similar to our work, PRISMA [16] used bytewise 3-gram vectors to represent binary messages and token vectors to represent text messages. However, because a single telnet message may carry both binary and text contents, we need a better embedding method to adapt to the complexity of telnet protocol. We empirically categorize payload bytes into three types: alphanumeric, symbolic (plus punctuation and spaces), and unprintable. As we assume that the type of each byte and its collocation imply semantic information, we split the request log at positions where two contiguous bytes are different types. We consider these tokens as minimum semantic units of a request log and build a Bag of Word (BoW) vector to represent its basic semantics in the feature space. This method generates a token that consists of only one type of byte and enables the extraction of information from all bytes.

Besides BoW vectors, we also use n-gram vectors to highlight the replacement of variable tokens in different loaders. Assigning a high value to n may result in computational overhead, so we choose 2-gram and 3-gram of tokens to capture the variability while limiting the scale of feature vectors. We join these three vectors to generate a feature vector for every request log.

In order to measure the semantic distance between request logs, the dissimilarity metric should reflect the existence, repetition, and collocation of tokens and n-gram features based on the proposed feature vector. We choose the Euclidean distance in our experiment, because the commonly used cosine distance may not reflect the repetition of tokens.

C. Agglomerative Clustering

To cluster similar loaders yet demonstrate their inter-cluster similarities, we use agglomerative clustering to build hierarchy clusterings bottom-top, based on dissimilarity metrics. The clustering process starts from single-element clusters corresponding to every request log. In each iteration, the algorithm searches for two clusters having the minimum inter-cluster distance based on the distance metric (also known as the dissimilarity metric in this work) and a linkage criterion. The algorithm hierarchically merges two clusters in each iteration until only one is left.

Here, we describe the hierarchical clusters as an inverted binary tree, on which the leaves at the bottom correspond to request logs, while a trunk node refers to an agglomerative cluster of attached leaves. The height of a cluster node refers to the inter-cluster distance of its two sub-clusters, which also indicates the discrepancy of contained elements. We denote agglomerative clusters as $C \in \mathbb{N}$, where $\mathbb{N}$ denotes the full set of them. Every cluster C can be further split into two sub-clusters $\{C_1, C_2\}$ or merged into a super-cluster C^S. Cutting the tree at a given height $\mathcal{T}$ will produce a preliminary partitioning $\mathbb{P} \subset \mathbb{N}$ at a selected precision. To make generated clusters cohesive yet discrete from each other, we determine the value of $\mathcal{T}$ based on the shape of the tree as discussed later. In this work, we use the `ward` [17] function offered by Scikit-learn, as the linkage criterion minimizes the variance of the merged clusters.

D. Pattern Extraction

We apply the Smith-Waterman algorithm [18] from leaves to the root node to get "templates" for every cluster on the agglomerative binary tree and identify shared patterns of sibling loaders or clusters out of their request logs.

We use the tokenized sequence in the section III-B to align two request logs. As depicted in Fig. 2, the $align(A_{C1}, A_{C2})$ operation leverages the Smith-Waterman algorithm to scan two token sequences from head to tail. This operation aligns identical tokens at the same position and adds placeholders (shadow cells in Fig. 2) to replace the mismatched ones, allowing identical tokens to align. We finally get a "template" of two clusters indicating the shared pattern of tokens. For any cluster $C \in \mathbb{N}$, the corresponding template A_C is generated recursively based on the templates of its sub-clusters A_{C1} and A_{C2}. Every node on the agglomerative tree will get a "template" representing the common pattern of its elements.

E. Clustering Refinement

As the unique $\mathcal{T}$ value may not fit all branches on the tree, the preliminary partitioning $\mathbb{P}$ is far from being taken as the final class definition. Starting from nodes in $\mathbb{P}$, we examine corresponding templates to calibrate the family definition by

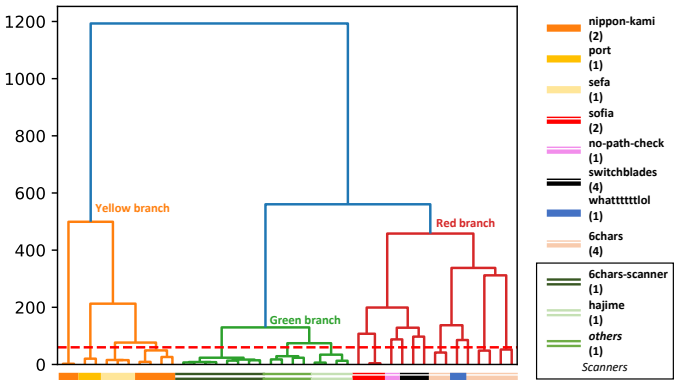

Fig. 3. Clustering dendrogram of the agglomerative clustering. Only the top five layers are shown. The red dashed line denotes $\mathcal{T} = 60$. The colored bars indicate the family definition on the agglomerative tree. For every family, we note the count of member clusters/samples in the legend on the right side.

TABLE II
INDEX TABLE OF LOADER FUNCTIONS DISCUSSED IN SECTION IV-C

Loader families	Intialize		Get work dir		Monopolize	Test env			Load & run malware				
	Enable shell	Check env	Get mounts	Scan path		Test copy	Check arch	Check dler	Drop malware	Pre-exec	Exec	Post-exec
Nippon-kami	Partly volatile	A_1	$B_{1\text{-}1}$	$B_{1\text{-}2}$	C_1	$D_{1\text{-}1}$	$D_{1\text{-}2}$	$D_{1\text{-}3}$	$E_{1\text{-}1}$	$E_{1\text{-}2}$	$E_{1\text{-}3}$	$E_{1\text{-}4}$
SEFA		A_2										
Port		A_1							E_2			
No-path-check	-						D_2		$E_{1\text{-}1}$	$E_{1\text{-}2}$	$E_{1\text{-}3}$	$E_{1\text{-}4}$
SwitchBlades	Volatile	-	-	B_2	C_2	$D_{1\text{-}1}$						
Sofia	Long	-	-		C_3	-	-	-	-	$D_{1\text{-}1}$	-	-
6-chars	-	A_3	-	B_3	C_4	-	-	-	$E_{3\text{-}1}$	-	$E_{3\text{-}2}$	-
whatttttlol	Fixed	A_4	-	B_4	-	-	-	-	$E_{4\text{-}1}$			$E_{4\text{-}2}$

* The capital letters A-E correspond to the 5 categories of loader functions listed in the upper header. Their different implementations are indicated by the first digit of their subscripts, while the second digit indicates sub-commands called for the corresponding function.

evaluating if a cluster should be kept, merged, or further split. Here, we empirically configure some criteria for accepting or denying a merged cluster:

- While identical commands are critical in evaluating the similarity, their arguments and arrangement are also important factors that we should concern about.
- For complex statements, the syntax structure is more important than its component commands to evaluate the similarity of two templates.
- We ignore the variation of self-identification tokens unless it appears in different commands or arguments.

IV. Data Analysis

In this section, we discuss the functions and behavior of active loaders based on the aforementioned methods and make a conclusion about their homologies.

A. Captured Dataset

The following analysis is based on captured request logs from November 14 to December 31 in 2021. To reduce the scale of the dataset, we take no more than 20 request logs for each host and selected 4,855 out of over 3 million captured items. As this work focuses on the function of active loaders instead of their deployment, we drop duplicates and got 481 valid items. This dataset generates 895 tokens, 2,451 2-gram terms, and 4,737 3-gram terms, finally composing feature vectors of 8,083 dimensions.

B. Family Definition

In this step, we leverage agglomerative clustering to define several families of bot loaders based on the collected dataset.

1) Clustering overview: The agglomerative clustering algorithm generates a tree with a height of 1193.07, whose dendrogram is depicted in 3. While the tree is relatively tall, most of the branches are at a height below 200. Minority branches at a higher height manifest significant discrepancies in samples in the corresponding clusters. Based on the method in Section III-D, we recursively generate templates for 480 non-singleton clusters to describe their common behaviors.

2) Threshold selection: According to the dendrogram in Fig. 3, when trying to merge two sibling clusters with a distance over 100, the intra-cluster discrepancy of the merged cluster will increase greatly compared to the original ones, which runs counter to our expectation of clustering results. Based on this observation, we empirically set $\mathcal{T} = 60$ with reasonable margin. The $\mathcal{T}$ value partitioned all request logs into 19 clusters. As Griffioen's work [4] only intensively investigated 14 active botnets, we regard the partitioning as reasonable to reflect the situation of active loaders.

3) Family definition: Based on the extracted templates, we evaluated these clusters based on the aforementioned criteria in section III-E, and finally identified several families out of the dataset. We traverse their sub-clusters and pick some representative tokens as their name, which may not follow the common naming rules. We listed these families and indexed their functions in Table II.

The yellow branch in Fig. 3 consists of 3 very similar families: *Nippon-kami*, *SEFA*, and *Port*. As the control group data from Anna-senpai's loader is all located on the *Nippon-kami* branch, in our following analysis we treat *Nippon-kami* as an alias of the original Mirai loader family.

We also identified five "red families" in Fig. 3 that are significantly different from the aforementioned "yellow families":

- *No-path-check* removes every command prior to checking the architecture and simply uses the default working directory of the logged-in user;
- *SwitchBlades* derives the framework of *Nippon-kami*, but uses a different implementation to detect writable directory, acting similar to *Sofia*;
- *Sofia* bases its intrusion toolkit on a simplified workflow, using a long initial command list and implementing a different method to detect writable directories;
- *6-chars* generates 6 random escaped characters to check the shell environment for every session, acting differently from every other family;
- *"whatttttlol"* does not share any pattern with other families. It runs a fixed command list and downloads multiple scripts named "`whatttttlol*.sh`" to load

$B_{1\text{-}2}$

```
busybox echo -e '\\x6b\\x61\\x6d\\x69/proc' > /proc/.nippon;
busybox cat /proc/.nippon;
busybox rm /proc/.nippon
```

B_2 *and* B_3

```
>/var/tmp/.file && cd /var/tmp/
```

$D_{1\text{-}1}$

```
/bin/busybox cp /bin/echo sefaexecbi; >sefaexecbi; /bin/
    busybox chmod 777 sefaexecbi;
```

$D_{1\text{-}2}$

```
/bin/busybox cat /bin/echo
```

D_2

```
/bin/busybox cat /bin/busybox || while read i; do echo $i;
    done < /bin/busybox
```

Fig. 4. Sample codes of the "Get working directory" function (B_*) and the "Test environment" function (D_*) denoted by indexes in Table II.

the bots.

4) Clustering of scanners: The samples on the green branch are very different from the other ones. As they only conduct quick probes and do not run any downloading commands, we assume that these logs are related to scanning campaigns. Among these scanner logs, we first identify the *6-chars-scanner* family that generates 6 random escaped characters in their scanning conversation. We also identify the *hajime* family related to our control group sample generated by a Hajime bot, as well as the *others* family having no particular patterns.

C. Behavior Patterns

In this section, we interpret the Table II vertically to make a comprehensive comparison about their shared pattern. We denote all functions of loaders by alphanumeric indexes.

1) Initialize: At the beginning of the intrusion, the loader injects initializing commands to enable the shell interface and checks the environment. As yellow families share the same codebase, their initialize command lists are very similar. They run `ps` command to check suspicious processes in the environment (A_1). The SEFA loader modifies the victim's hostname to `SEFA_ID:<4-digit numbers>` (A_2) to identify bots in the botnet. Sofia removed all checking commands but extended the initialize command list. While whatttttttlol holds a fixed command list, it runs `ls /home` to scan files in the directory (A_3). 6-chars only checks `wget` in this step (A_4).

2) Get working directory: Most of the loaders require a writable directory to temporarily drop the executable. Yellow families scan mounted filesystems ($B_{1\text{-}1}$) and create some files($B_{1\text{-}2}$) to check their writing privileges. SwitchBlades, Sofia, and 6-chars use a simplified statement (B_2 and B_3 in Fig. 4) to test writable directories in their own hard-coded lists. While SwitchBlades and Sofia use returns to assemble these element statements (B_2), the 6-chars family uses semicolons (B_3) which makes a slight difference. A variant of 6-chars runs this step twice, which shows a difference in the request logs. Whatttttttlol uses a simple "`||`" (or) statement to join multiple `cd <directory>` commands (B_4). This statement changes the working directory to the first available one in the hard-coded list, regardless of its writable privilege.

3) Monopolize: Most loaders will try eliminating competitors by deleting certain files stored in a built-in list. The yellow families use `.sh .t .human` (C_1), Sofia uses `.file .cowbot.bin retrieve cowffxxna` (C_3), and 6-chars uses `.i` only (C_4). SwitchBlades tries to delete two files while the lists are unstable among different variants (C_2).

4) Test environment: In this step, the loaders probe the CPU architecture and the available downloaders to decide how to load a bot. Yellow families tests `cp` command($D_{1\text{-}1}$), prints `/bin/echo`($D_{1\text{-}2}$), and then test `wget` and `tftp` commands ($D_{1\text{-}3}$) in this step. As the CPU architecture can be obtained by parsing any executable on the device, no-path-check and Sofia prints `/bin/busybox` to obtain the same information (D_2). In case of the `cat` command is unavailable, they also use a shell-based `while read` statement to print the file. Sample codes are displayed in Fig. 4.

5) Drop & run malware: In this step, loaders `cd` to the selected working directory and drop bot clients via a tested downloader. If neither `wget` nor `tftp` is available, most Mirai-based families will run a fallback command that loads the whole file with `echo` command and launches a stdout redirect statement ($E_{1\text{-}*}$). 6-chars leverages an "`||`" (or) statement to call multiple commands sequentially ($E_{3\text{-}*}$) until a command succeeds. Whatttttttlol calls multiple commands sequentially to download and run 4 scripts, and then it deletes them all after the execution to clean the trace ($E_{4\text{-}*}$). In this step, Port and Sofia do not seem to download any executable. Instead, Port calls `openssl` for an unknown reason (E_2), while Sofia only checks writable privilege in the current directory ($D_{1\text{-}1}$).

V. Discussion

Based on the agglomerative tree and the discussion about behavior patterns, we draw a dendrogram (Fig. 5) to demonstrate the variation of intrusion functions.

A. Variation of Loaders

Although we treat *nippon-kami* family as the direct descendant of the original Mirai loader, we found that other families inherit its intrusion workflow, but modify some components to adapt to different environments and situations. The directory detector is frequently modified or rebuilt to fit heterogeneous filesystem structures on victim devices, while the cleaning commands used for monopolizing the infected device also vary according to the malware family. Some families made significant changes to the original code base to simplify the workflow (*No-path-check* and *SwitchBlades*) or rebuild the toolkit (*Sofia*). Some independent families also implemented their toolkits to load malware.

While conventional taxonomy research overlooked the variation and evolution of bot loaders, this experiment reveals that Mirai original ideas and codebase are still contributing to new spawning variants. Our server-side perspective highlights how the infection mechanisms of these bots operate through telnet and how they are suitable to run on different environments.

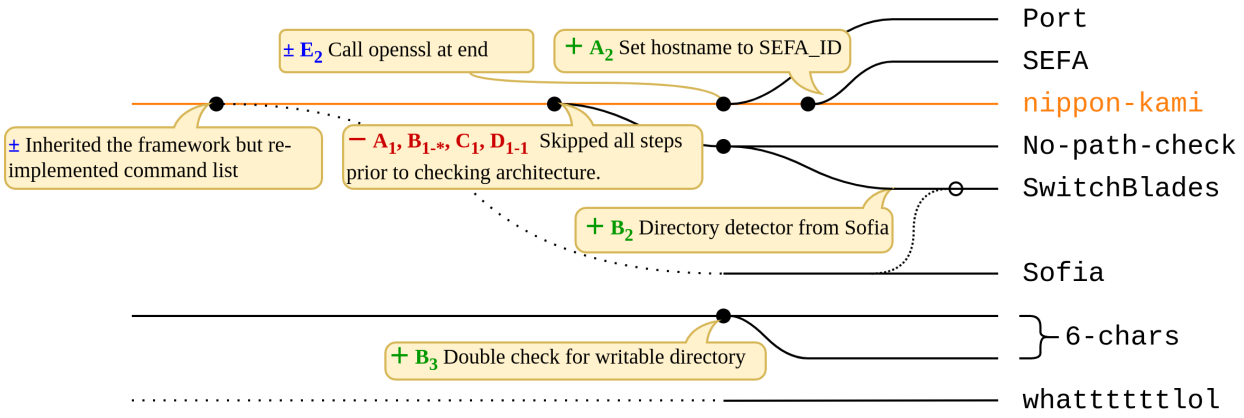

Fig. 5. Dendrogram of loader families. Horizon lines depict identified families, while the yellow one highlights the *nippon-kami* released by Anna-Senpai (the author of Mirai). In the description of a connecting line between two families, green "+" indicates adding functions, red "-" indicates removing functions, and blue "±" indicates modifications. All the functions are denoted by indexes in Table II.

B. Comparison with Other Studies

According to Cozzi [5] and Wang [7], the evolution of bot clients focused on updating their scanners, attackers, persistence techniques, and anti-detection techniques. Our work demonstrated the distinct motivation of loaders' and bots' evolution. Compared to Torabi's work [14] and Tabari's work [15], our work further quantified the similarity of families and identified the lineage of loaders beyond simple comparisons of string patterns, which contributes to understanding the evolution of botnet malware from new perspectives.

As noted by Wang [7] and Dang [9], a growing number of botnets are exploiting victims by means of fileless attacks. We noticed the *Port* family replaced *Nippon-kami*'s loading tool with a fileless attack command, which broke Mirai convention behaviour of infecting victims by downloading executables. Our lineage study sheds some light on the provenance of fileless attack toolkits and helps to understand how botmasters develop new attack vectors starting from the original Mirai codebase. In turn, this knowledge can contribute to improve the efficiency of defense strategies against new variants.

VI. Conclusion

In this paper, we analysed telnet request logs captured with ad-hoc honeypots, and we investigated functions and similarities of various infection loaders. Our data allowed us to define 8 different families and draw a dendrogram of their lineage and evolution, demonstrating the importance of understanding loaders' evolution and variation. The experiment highlighted the evolution of IoT botnets on the server side, providing a server-side view of botnets evolution and a novel behavior-based taxonomy of bot loaders.

References

[1] Anna-senpai, "[FREE] World's Largest Net:Mirai Botnet, Client, Echo Loader, CNC source code release," https://hackforums.net/showthread.php?tid=5420472, Sep. 2016.

[2] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, "Understanding the Mirai Botnet," in *USENIX Security 17*, 2017.

[3] S. Herwig, K. Harvey, G. Hughey, R. Roberts, and D. Levin, "Measurement and Analysis of Hajime, a Peer-to-peer IoT Botnet," in *NDSS '19*, 2019.

[4] H. Griffioen and C. Doerr, "Examining Mirai's Battle over the Internet of Things," in *CCS' 20*, Oct. 2020.

[5] E. Cozzi, P.-A. Vervier, M. Dell'Amico, Y. Shen, L. Bilge, and D. Balzarotti, "The Tangled Genealogy of IoT Malware," in *ACSAC '20*, Dec. 2020.

[6] E. Downing, Y. Mirsky, K. Park, and W. Lee, "{DeepReflect}: Discovering Malicious Functionality through Binary Reconstruction," in *USENIX Security 21*, 2021.

[7] H. Wang, W. Zhang, H. He, P. Liu, D. X. Luo, Y. Liu, J. Jiang, Y. Li, X. Zhang, W. Liu, R. Zhang, and X. Lan, "An Evolutionary Study of IoT Malware," *IEEE Internet of Things Journal*, Oct. 2021.

[8] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow, "IoTPOT: Analysing the Rise of IoT Compromises," in *WOOT '15*, Aug. 2015.

[9] F. Dang, Z. Li, Y. Liu, E. Zhai, Q. A. Chen, T. Xu, Y. Chen, and J. Yang, "Understanding Fileless Attacks on Linux-based IoT Devices with HoneyCloud," in *Mobisys '19*, Jun. 2019.

[10] O. Alrawi, C. Lever, K. Valakuzhy, R. Court, K. Snow, F. Monrose, and M. Antonakakis, "The Circle Of Life: A {Large-Scale} Study of The {IoT} Malware Lifecycle," in *USENIX Security 21*, 2021.

[11] "VirusTotal," https://www.virustotal.com/.

[12] "YARA," https://virustotal.github.io/yara/.

[13] B. Lingenfelter, I. Vakilinia, and S. Sengupta, "Analyzing Variation Among IoT Botnets Using Medium Interaction Honeypots," in *CCWC '20*, Jan. 2020.

[14] S. Torabi, M. Dib, E. Bou-Harb, C. Assi, and M. Debbabi, "A Strings-Based Similarity Analysis Approach for Characterizing IoT Malware and Inferring Their Underlying Relationships," *IEEE Networking Letters*, vol. 3, no. 3, pp. 161–165, Sep. 2021.

[15] A. Z. Tabari, X. Ou, and A. Singhal, "What are Attackers after on IoT Devices? An approach based on a multi-phased multi-faceted IoT honeypot ecosystem and data clustering," *arXiv:2112.10974 [cs]*, 2021.

[16] T. Krueger, H. Gascon, N. Krämer, and K. Rieck, "Learning stateful models for network honeypots," in *AISec '12*, 2012.

[17] J. H. Ward, "Hierarchical Grouping to Optimize an Objective Function," *Journal of the American Statistical Association*, vol. 58, no. 301, pp. 236–244, Mar. 1963.

[18] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, Mar. 1981.