Detection of Mirai by Syntactic and Semantic Analysis

Najah Ben Said, Fabrizio Biondi, Vesselin Bontchev, Olivier Decourbe,
Thomas Given-Wilson, Axel Legay, Jean Quilbeuf

Detection of Mirai by Syntactic and Semantic Analysis

Najah Ben Said*, Fabrizio Biondi†, Vesselin Bontchev‡, Olivier Decourbe*,
Thomas Given-Wilson*, Axel Legay*, and Jean Quilbeuf*
*Inria, France
Email: {najah.ben-said,olivier.decourbe,thomas.given-wilson,axel.legay,jean.quilbeuf}@inria.fr
†CéntraleSupelec, France
Email: fabrizio.biondi@inria.fr
‡National Laboratory of Computer Virology, Bulgarian Academy of Sciences, Bulgaria
Email: vbontchev@yahoo.com

***Abstract*—The largest DDoS attacks in history have been executed by devices controlled by the Mirai botnet trojan. To prevent Mirai from spreading, this paper presents and evaluates techniques to classify binary samples as Mirai based on their syntactic and semantic properties. Syntactic malware detection is shown to have a good detection rate and no false positives, but to be very easy to circumvent. Semantic malware detection is resistant to simple obfuscation and has better detection rate than syntactic detection, while keeping false positives to zero. This paper demonstrates these results, and concludes by showing how to combine syntactic and semantic analysis techniques for the detection of Mirai.**

1. Introduction

On any Internet-exposed service, it is fundamental that the service is able to handle connections and requests to perform its function. However, the infrastructure supporting the service necessarily has a limited capacity to handle requests. An attack able to prevent a service from performing its function is known as a Denial of Service (DoS) attack [28].

When a single device is performing a DoS attack, it is common for the service to blacklist the attacker's address to prevent the attack from succeeding. Thus Distributed Denial of Service (DDoS) attacks use thousands or even hundreds of thousands of devices in parallel to execute DoS attacks [11]. Typically, large networks of devices remotely controlled by an attacker are used to execute DDoS attacks. Such networks are known as *botnets*, and are usually built by infecting unprotected devices with a trojan that takes control of the device and waits for commands from the controller of the botnet [23].

DDoS attacks are among the most difficult attacks to protect a system against [11]. When a botnet of hundreds of thousands of nodes attacks the same target, it is impossible for the target to distinguish legitimate traffic from attack traffic, and the service becomes unusable. The largest DDoS attacks to date have reached 620 Gbps and 1Tbps of network traffic generated by the attack [17], [20]. Both of these attacks have been generated by botnets created with the same malware: the Mirai botnet trojan. Due to its proven effectiveness in collapsing network infrastructure, Mirai can be considered as a powerful cyberweapon that is outside the control of any government. Hence, stopping it is an important issue for the stability and reliability of the global Internet infrastructure.

The goal of this paper is to study Mirai and to determine effective methods to stop the spread of Mirai. Since Mirai infects IoT devices that are assumed to have limited computational power, the device itself cannot be relied upon to have effective antivirus capabilities. In fact, even changing the default password of the device would be enough to stop the basic versions of Mirai, but the extensiveness of the botnet shows that this is not happening in practice.

The responsibility for detecting and stopping Mirai thus falls to the network firewall to be able to detect Mirai samples in transit. That is, infrastructure devices should be able to determine whether a transiting binary file is Mirai.

Malware detection is commonly based on *signatures* of known binaries, where some characteristic information of the binary (the signature) is extracted from the binary and compared to signatures of known malware to find if there is a correspondence.

Signature-based malware detection normally proceeds as follows. First, the analyst obtains a number of samples of the malware they want to detect. Second, the analyst also obtains a database of cleanware that is not supposed to be detected as malware. Third, the analyst extracts the signatures of each malware and cleanware sample and tries to find common characteristics of the malware signatures that do not appear in the cleanware signatures. Fourth, the characteristics that distinguish the malware from the cleanware can be used to build a signature for the malware. Fifth, this signature can be used to classify a new sample to determine whether it is malicious or clean.

The fourth and fifth steps are sometimes implemented automatically using a supervised classification machine learning algorithm. The algorithm is trained over features of the samples of the malware and cleanware, and becomes able to automatically classify samples as malicious or clean. Note that malware detection based on anomaly detection

also exists, but it is not discussed here since it's not as effective in detecting a specific, known family of malware.

This paper distinguishes between *syntactic* and *semantic* malware detection according to whether the signatures used are based on syntactic or semantic properties of the binary, respectively.

Syntactic properties (also known as static properties) are those that are easy and computationally inexpensive to extract, in particular they do not require any analysis of the program behaviour or simulation of computation. Syntactic properties of a binary include: its size, its sections, the entropy of the whole binary or some of its sections, the value of some specific bytes (e.g. its magic number), and its strings. However, it is also easy for malware creators to modify such properties using obfuscation techniques to hinder signature extraction. This paper uses the state-of-the-art YARA tool [3] for malware detection based on syntactic signatures. YARA syntactic signatures, known as *rules*, are commonly used to relay information about the detection of malware binaries. While automated rule generation mechanisms exist [7], [9], rules are mostly hand-written or refined by human analysts. YARA uses an efficient pattern matching algorithm to determine whether a binary corresponds to any of the rules in a YARA rule database. The authors of YARA rules are responsible for ensuring efficient matching, high detection, and low false positive rate. This work uses two different YARA rule databases for Mirai detection, maintained by Virustotal and by Florian Roth. The rules are mostly based on the strings that are visible in the binary. Full details are given in Section 4.

Semantic properties (also known as behavioral or dynamic properties) are more expensive than static properties to extract, since they require some kind of behavioral analysis of the binary. Semantic properties include: its control flow, its interaction with the system it is running on, and its network communications. However, semantic properties are far harder for malware creators to obfuscate than syntactic properties, since they rely on the actual behavior of the malware. This paper uses the interactions with the system as summarized in a System Call Dependency Graph (SCDG). The SCDG maps all interactions of the binary with the system (in the form of its system calls) as the nodes in a graph, and connects nodes that are logically dependent on each other (e.g. creating a file, writing on that file, and closing that file). This paper uses symbolic analysis based on the Angr tool [34] to extract SCDGs from binaries to be used as semantic signatures. The classification is based on machine learning, using the gSpan common subgraph algorithm to extract relevant fragments of the SCDGs characterizing Mirai behavior and using distance metrics to determine whether the SCDG of a sample binary is similar enough to the SCDGs of Mirai to classify the sample as Mirai. Full details are given in Section 5.

Due to the different strengths and weaknesses of the syntactic and semantic approaches, it is reasonable to try and combine both to obtain a technique more effective than either one alone. This paper also considers how to construct such a combined syntactic and semantic system and empower the system with whitelisting to both: reduce false positives, and to increase efficiency of the system. Full details are given in Section 6.

Note that while the techniques of this paper are focused on Mirai, they can be adapted to any other malware family and extended to multi-family detection and classification.

This paper provides the following contributions.

1) Describing the capabilities of the Mirai botnet trojan, including its infection and replication methods and the trojan's common behavior.
2) Evaluating the effectiveness of syntactic malware detection based on the YARA tool on a database of 516 Mirai binaries and 6438 clean binaries, showing that YARA obtains an $F_{0.5}$-score of 98.69% with zero false positives. The choice to rank by $F_{0.5}$ score was to give greater weight to false positives, since these are considered more detrimental to good performance in malware classification.
3) Showing how simple string obfuscation techniques reduce YARA's accuracy to zero.
4) Evaluating the effectiveness of semantic malware detection based on SCDGs and the gSpan algorithm on the same database, obtaining an $F_{0.5}$-score of 99.78% again with zero false positives.
5) Discussing the limitations of the semantic technique.
6) Describing a combined syntactic and semantic detection technique that includes feedback to improve effectiveness and efficiency.

The structure of the paper is as follows. Section 2 describes Mirai. Section 3 discusses the database of samples used for the analysis. Section 4 describes the results of using syntactic analysis based on YARA for Mirai detection. Section 5 describes the results of using semantic analysis based on SCDGs and the gSpan algorithm for Mirai detection. Section 6 describes how to combine the syntactic and semantic approaches to obtain better results than either technique. Section 7 concludes the paper.

2. The Mirai Botnet

This section describes the Mirai malware family analyzed in this paper.

2.1. Timeline

The initial activity of Mirai can be traced to August 2016, generating distributed denial of service (DDoS) attacks based on generic routing encapsulation (GRE) floods peaking at 280 Gbps and 130 Mpps [13]. Also in August 2016, independent malware researchers at MalwareMustDie! published a full analysis of Mirai's capabilities and attack vectors, identifying Mirai as an evolution of malware previously known as GayFgt, Torlus, Lizkebab, Bash0day, Bashdoor, or BashLite [24].

The attention on Mirai increased on the 20th to 22nd of September 2016 when Mirai was used for a 620 Gbps DDoS attack by approximately 120,000 devices on the website KrebsOnSecurity [20], and a 1 Tbps DDoS attack by approximately 150,000 devices on French internet service and hosting provider OVH [17] leading to worldwide scrutiny. The attacks were peculiar by being able to reach such a large volume without using any DNS reflection, while instead relying on a large network of IoT devices, mostly closed-circuit cameras (CCTV), routers, and digital video recorders (DVRs).

On the 30th of September 2016 Mirai's author, nicknamed "Anna-senpai", released the source code of Mirai, allowing for deeper study and replication [21]. A mapping of Mirai botnets was published on the 3rd of October 2016 by security researcher MalwareTech [25]. On the 21st of October 2016, a Mirai botnet was used for a series of DDoS attacks against DNS provider Dyn, impacting the availability of many websites including PayPal, Twitter, Reddit, GitHub, Amazon, Netflix, Spotify, and Runescape [31]. On the 3rd of November 2016 a 500 Gbps DDoS attack by a Mirai botnet against a mobile provider in Liberia has been reported by security researcher Kevin Beaumont [4]; however, the attack was nullified by DDoS mitigation with minimal impact on the users [19]. A variant of Mirai with SOAP exploitation and additional propagation through the TR-064 and TR-069 protocols caused an outage for over 900,000 Deutsche Telekom customers on the 28th and 29th of November 2016 [10]. Another variant was used for a 54-hour DDoS attack on a US college starting on the 28th of February 2017 [5].

2.2. Capabilities

Mirai malware infects computers (mostly devices like CCTVs, DVRs, etc.) running the Linux operating system and combines these computers into a large botnet. This botnet is usually then used for DDoS attacks. Thus, Mirai has two main functions - infection and attack. The original Mirai consists of 3 distinct programs - bot, downloader, and server.

Taxonomically, we note that Mirai is neither a virus nor a worm. A virus is a program capable of replicating itself, and a worm is a virus explicitly using the network to replicate itself. In the particular case of Mirai, neither of the three parts is capable of replicating itself, nor does the set of three parts replicate itself from one computer to another.

Nevertheless, the botnet expands itself by coopting new vulnerable computers, because one of the parts (the server) copies another part (the dowloader) to these computers and the downloader copies the bot from the server to the computer it has infected.

2.2.1. The Bot. The Mirai *bot* is the software that resides on the infected devices. The bot has two primary functions: scan the Internet for vulnerable hosts and communicate them to the server, and conduct DDoS attacks of the kind and against targets specified to the bot by the server.

Immediately after the bot is started on the newly infected device the bot: deletes itself from the file system, modifies its process name to a random alphanumeric string, and wipes its command-line arguments. These are all performed in order to hide the presence of the bot. The bot then kills the processes that listen to TCP ports 22 (SSH), 23 (Telnet) and 80 (HTTP), in order to prevent other bots from connecting to the infected device.

The bot also examines the running processes (by reading the files in the directory /proc/) and attempts to kill various other "competing" bots, if they are found to be present. In particular, the bot looks for the strings "REPORT %S:%S", "HTTPFLOOD", "LOLNOGTFO" (corresponding to various variants of the QBot malware), "\X58\X4D\X4E\X4E\X43\X50\X46\X22" (anything packed with the executable packer UPX), "ZOLLARD" (the Zollard bot), as well as for any processes that have ".anime" in their name. Any such process, if found, is swiftly terminated.

The Mirai bot does not use any mechanism for persistence. Therefore, it can be removed by just rebooting the infected device. However, given the prevalence of Mirai-infected machines actively scanning the Internet, any clean vulnerable device will be re-infected within seconds, if connected directly to the Internet (i.e., not behind a firewall that blocks incoming telnet connections).

Scanning. The bot generates random IP addresses and checks whether vulnerable computers reside at these addresses. The IP addresses are generated using the Xorshift128 pseudo-random number generator [26]. The bot explicitly ignores IP addresses belonging to some specific subnets, listed in Table 6 in the Appendix. If a telnet connection (TCP port 23) to a generated IP address is successful, the bot tries to log in, using one of 60 username/password pairs. The full list is given as Table 7 in the Appendix.

The bot verifies that it has succeeded to log in by issuing the following commands:

```
enable
system
shell
sh
/bin/busybox MIRAI
```

The first four commands attempt to start a shell in various environments. The last command invokes a multi-purpose system program (busybox) with an invalid argument ("MIRAI"). The reason for this is because the system prompts vary in the different environments, so the bot uses the response of busybox when an invalid argument has been passed to it to determine that the system has reacted to its commands and, therefore, it has successfully logged in.

If the bot determines that it has successfully logged into a vulnerable machine, it contacts the server and communicates the IP address of the machine, the TCP port that the bot opened on the machine, and the username/password pair used for successful login.

DDoS Attacks. The bot also listens for commands from its server, instructing it to perform various attacks against specified targets. The commands specify the type of DDoS attack, the IP/subnet of the target, and duration of the attack. The following types of DDoS attacks are supported:

- Simple UDP flood
- UDP VSE (Valve Source Engine) flood
- DNS flood
- SYN flood
- ACK flood
- STOMP flood
- GRE IP flood
- GRE ETH (Ethernet) flood
- UDP flood optimized for speed
- HTTP/GET flood

The commands can fine-tune the attacks by varying different parameters of the packets used in the floods - packet size, TTL (time-to-live), source IP, source port, destination port, etc.

2.2.2. The Downloader. The Mirai *downloader* is a very small program (about a kilobyte), whose only function is to download the version of the bot for the appropriate platform from the server and to transfer control to the bot. Strictly speaking, the downloader is not necessary - if the server could send the downloader to the vulnerable device, the server could just as easily send the bot itself. Indeed, many newer versions of Mirai do not use a downloader and consist of only two substantially different program parts.

2.2.3. The Server. The Mirai *server* receives from the bots the results of their scanning of vulnerable hosts as (IP, port, username, password) tuples. For each tuple the server proceeds to log into the vulnerable machine.

Once logged in, the server executes the following commands:

```
enable
shell
sh
/bin/busybox ECCHI
/bin/busybox ps; /bin/busybox ECCHI
/bin/busybox cat /proc/mounts; \\
 /bin/busybox ECCHI
```

As with the bot, the response from passing an invalid argument to busybox is used to determine that the machine has finished processing the commands issued by the server and has finished sending its responses to them. The contents of `/proc/mounts` lists the available mounted file systems. Each one of them is tested to check whether it is writable by creating a short text file on it, examining its contents, and removing it:

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

If such a writable directory is found, the server switches to it and creates an executable file there:

```
cd <DIR>
/bin/busybox cp /bin/echo dvrHelper;\\
 >dvrHelper; /bin/busybox chmod 777 \\
 dvrHelper; /bin/busybox ECCHI
```

The server then examines the contents of the file `/bin/echo`, in order to determine the CPU platform of the victim. The original Mirai variant supports 9 different platforms (ARM, ARM7, Motorola 68000, MIPS, Mipsel, SH4, SPARC, PowerPC and Intel x86), although some variants support up to 13 different platforms (additionally ARM5N, Intel 686, PowerPC Floating Point, 64-bit Intel x86). If the platform is ARM, the malware also uses the contents of the file `/proc/cpuinfo` to determine exactly what kind of ARM platform this is:

```
/bin/busybox cat /bin/echo
/bin/busybox ECCHI
cat /proc/cpuinfo; /bin/busybox ECCHI
```

The server then checks for the presence of the programs `wget` or `tftp`. If one of them is found, it is used to download the downloader from the repository and to launch the downloader in order to download the bot. Only the image of the downloader and the bot for the correct CPU platform is downloaded from the repository.

Theoretically, Mirai can have its parts distributed among 3 different machines: the vulnerable device running the bot, the server communicating with the bots and performing the infection, and the repository from where the server takes the components (downloader and bot) to upload to the infected devices. In practice, however, the server and the repository often reside on one and the same machine, usually a virtual machine on some cloud service provider. The address of the server is hard-coded (in encoded form) in the body of the bots, so when moving to a new server (e.g., because the old one has been shut down due to abuse complaints) the bot has to be re-compiled, producing at least a slightly different new variant. This (along with the free availability of the source code) is why there are so many Mirai variants.

3. Sample Set

This section describes the set of Mirai and clean samples used for the analysis in the rest of the paper.

The Mirai samples were collected using a honeypot for IoT malware from the 6th of April 2017 to the 14th of August 2017. During this time, more than 500 unique variants of malware were captured, each variant being available in samples for at least 8 different architectures.

In this paper the honeypot samples were limited to those that were statically linked ELF binaries targeting x86 32-bit platforms. This left 526 binaries that were likely to be Mirai. A further 9 binaries were removed from the set: 3 because they were samples of different malware, and 6 because we were not able to ascertain what they were (either using any method in this paper, VirusTotal, or manual analysis), leaving 516 confirmed Mirai binaries.

The clean samples were taken from the static-get[1] distribution for obtaining statically linked ELF 32-bit binaries.

1. http://s.minos.io/

TABLE 1. NAME OF THE CAPTURED MIRAI SAMPLES AND THEIR FREQUENCY

File Name	Frequency
mirai.x86	53.68%
miraint.x86	25.97%
x86	5.43%
mirai.i686	3.10%
miraint.i686	2.91%
mirai.x86_686	1.55%
miraint.x86_686	1.16%
dlr.x86	1.16%
usb_bus.x86	0.97%
tveth.x86	0.78%
pein.x86	0.78%
mirai.i586	0.39%
81c4603681c46036.x86	0.19%
mm.x86	0.19%
lavertele.i686	0.19%
camili.x86	0.19%
lavertelent.x86	0.19%
lavertele.x86	0.19%
boot.x86	0.19%
gnome.x86	0.19%
rash.x86	0.19%
helper.x86	0.19%
lavertelent.i686	0.19%

This database allowed the collection of 6438 binaries confirmed to be clean.

The combined sample set thus includes 516 Mirai samples and 6438 clean samples, all of which are 32-bit statically linked ELF binaries. The statically linked restriction is due to Mirai being statically linked and so needing a similar property for the clean samples. The choice of 32-bit ELF is due to the clean samples being 32-bit.

The several variants of mirai collected have varying filenames. Table 1 list the filenames appearing as well as their frequencies. In Figure 1, we show the cumulative distribution function of the sizes of the captured mirai samples.

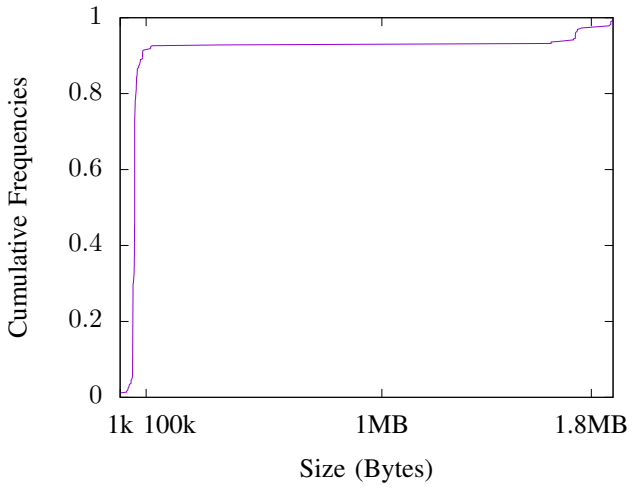

Figure 1. Cumulative distribution of the sizes of the captured mirai samples.

4. Mirai Detection by Syntactic Analysis

This section describes how the YARA tool has been used for detection of Mirai samples according to their syntactic properties. Section 4.1 describes syntactic malware classification in general, and Section 4.2 presents the YARA tool. Section 4.3 describes the methodology used to evaluate YARA as a tool to detect Mirai samples, and Section 4.4 presents the results of the evaluation. Finally, Section 4.5 discusses limitations of syntactic malware analysis and countermeasures that can be used against it.

4.1. Syntactic Malware Classification

This section presents malware detection based on syntactic properties. Basing malware detection and classification on automatically-extracted syntactic properties of malware like n-grams and strings is a consolidated approach with more than a decade of research [1], [18], [32], [33]. A similar approach is based on disassembling the binary file and then using the OpCodes of the function calls as signatures, using call frequencies [35], call sequences and permutations [16], or call sequence frequencies [37]. The SAFE [8] and SAVE [38] approaches follow similar principles. Good results on malware detection together with a feature relevance study have been recently published by Ahmadi et al. [2].

However, all syntactic techniques are limited by how easily the malware creators can deploy obfuscation techniques to hide strings, n-grams, and operations [27], [29], and the ones based on OpCodes also by the general difficulties of disassembling [6]. Generic deobfuscation approaches (e.g. [40]) are relatively recent and limited in their effectiveness against state-of-the-art obfuscation and packing.

4.2. The YARA Tool

YARA is used together with a *rule database* and a *target file* as follows.

```
yara <rule_file> <target_file>
```

YARA then returns the classification of the target file according to the rules in the rule file. Each rule in the database describe different conditions under which the target file can be classified by that rule. A fragment of one of the rules used in this paper for Mirai detection, written by Florian Roth, is shown in Figure 2.

In Figure 2, the fields `description`, `author`, `reference`, and `date` in section `meta` provide information about the rule itself. The rule includes the hash values of Mirai samples that were used to generate it, that have been removed from Figure 2 due to space reasons. Note that the hash values are included in the rule only for ease of reference, and are *not* used in the classification. In section `strings`, the fields starting with `$x` and `$s` represent strings known to be in Mirai samples. Finally, section `conditions` describes the conditions under which the target file should be classified as "`Mirai_Botnet_Malware`": the first two bytes of the

```
rule Mirai_Botnet_Malware {
    meta:
        description = "Detects Mirai Botnet
            Malware"
        author = "Florian Roth"
        reference = "Internal Research"
        date = "2016-10-04"
        // hashes omitted due to space reasons
    strings:
        $x1 = "POST /cdn-cgi/" fullword ascii
        $x2 = "/dev/misc/watchdog" fullword
            ascii
        $x3 = "/dev/watchdog" ascii
        $x4 = "\\POST /cdn-cgi/" fullword
            ascii
        $x5 = ".mdebug.abi32" fullword ascii

        $s1 = "LCOGQGPTGP" fullword ascii
        $s2 = "QUKLEKLUKVJOG" fullword ascii
        $s3 = "CFOKLKQVPCVMP" fullword ascii
        $s4 = "QWRGPTKQMP" fullword ascii
        $s5 = "HWCLVGAJ" fullword ascii
        $s6 = "NKQVGLKLE" fullword ascii
    condition:
        uint16(0) == 0x457f and filesize < 200
            KB and
        (( 1 of ($x*) and 1 of ($s*) ) or
            4 of ($s*))
}
```

Figure 2. A rule for Mirai detection.

TABLE 2. YARA MIRAI DETECTION RESULTS OBTAINED BY CHECKING THE SAMPLE SET WITH FLORIAN ROTH RULES (FR), VIRUS TOTAL RULES (VT) AND BOTH AT THE SAME TIME (BOTH).

Rule Set	FP Rate	FN Rate	Accuracy	Precision	$F_{0.5}$ score
FR	0.00%	7.56%	99.44%	100.00%	98.39%
VT	0.00%	47.87%	96.45%	100.00%	84.48%
Both	0.00%	6.20%	99.54%	100.00%	98.69%

file must correspond to `0x457f`, the file must be smaller than 200 KB, and the file must contain either a string from the `$x` list and a string from the `$s` list or four strings from the `$s` list.

4.3. Methodology

Syntactic analysis has here been performed using YARA version 3.6.3 on the entire sample set (see Section 3). Two different YARA rule sets have been used: the rules curated by VirusTotal (available at https://github.com/Yara-Rules/rules) and the rules curated by Florian Roth (available at https://github.com/Neo23x0/signature-base/tree/master/yara).

A sample has been classified as Mirai if it is classified as any variant of Mirai using either rule set.

4.4. Results

The Table 2 presents the detection results for each set of rules separately and for their combination. Yara does not produce any false positive, and no binary from the clean samples was detected as Mirai. However, about 6% of the Mirai binaries are not detected by these sets of rules.

4.5. Limitations and Countermeasures

The reliance of syntactic analysis on syntactic properties, such as file size and strings, makes it easy for malware creators to write malware that can circumvent syntactic analysis. As an example, consider again the YARA rule in Figure 2. It would be simple to circumvent this rule just by padding a Mirai binary so that its size is greater than 200 KB. More recent Mirai rules in the same file consider this by extending the size condition to `filesize < 5000KB`.

Even if a rule was modified to allow for larger Mirai samples, it could be circumvented by obfuscating the strings visible in the Mirai binary. Any packing technique that encrypts the binary and only unpacks it at runtime will change all the visible strings, making them impossible to detect by YARA or any other syntactic tool based on strings. However, packed binaries are easy to detect due to their increased entropy, and often antivirus software considers them suspicious and devotes further analysis to them.

In fact, obfuscating only the strings of the file is sufficient to circumvent YARA and much harder to detect by entropic analysis. Again in the rule in Figure 2, note how the strings of the list `$x` list are all strings that appear in the source code of the malware; for instance, strings `$x2` and `$x3` are declared on line 71 and 72 of `mirai/bot/main.c`:

```
if ((wfd = open("/dev/watchdog", 2)) != -1 ||
    (wfd = open("/dev/misc/watchdog", 2)) != -1)
```

However, if the strings are declared separately in the source code and then recomposed, e.g. by

```
strcpy(watch_dir1,"/dev");
strcpy(watch_dir2,"/dev/misc");
strncat(watch_dir1,"/watchdog",10);
strncat(watch_dir2,"/watchdog",10);
if ((wfd = open(watch_dir1, 2)) != -1 ||
    (wfd = open(watch_dir2, 2)) != -1)
```

then the strings do not appear in the binary and YARA does not detect them anymore.

It could be argued that the YARA rule could be updated to detect instead the strings `/dev`, `/dev/misc`, and `/watchdog`. However, another simple string obfuscation technique is to store two strings that when bit-XORed together produce the target strings, as in the following example.

```
char stra1[] = "\x57\x0b\x17\x13\x4b\x28
\x00\x2b\x0e\x01\x16\x0e\x0e";
char stra2[]  =  "xored_a_mirai";
char watch1[14];
char strb1[] ="\x42\x0d\x17\x17\x46\x32
\x07\x2c\x0c\x4d\x11\x14\x07\x00\x09\x10
```

```
\x0a\x03";
char strb2[]  =  "mirai_n_obfuscated";
char watch2[19];
for (i=0; i<13; i++){
  char temp = stra1[i] ^ stra2[i];
  watch1[i] = temp;}
watch1[i]='\0';
for (i=0; i<18; i++){
  char temp = strb1[i] ^ strb2[i];
  watch2[i] = temp;}
watch2[i]='\0';
if ((wfd = open(watch1, 2)) != -1 ||
    (wfd = open(watch2, 2)) != -1)
```

Since Mirai's actual strings can be produced randomly before or after compilation time, it is not possible to generate a YARA rule that will be able to detect them without having an exponential size in the size of the obfuscated strings.

Note that the rule in Figure 2 also detects a sample as Mirai if none of the `$x` strings are detected, as long as at least four of the `$s` strings are detected. Obfuscating the `$s` strings is also possible and also prevents YARA from detecting the obfuscated samples as Mirai. Since the obfuscation of the `$s` strings is less trivial than the obfuscation of the `$x` strings, this has not been detailed here to prevent malware creators from implementing them.

5. Mirai Detection by Semantic Analysis

This section describes how to extract semantic behavior from binary samples, generate semantic signatures, and classify unknown samples semantically. Semantic behavior of malware has been considered before [14], [15], [30], [36] with several approaches proposed. *System call dependency graphs* (SCDGs) prove to be an effective semantic representation of how a program interacts with the host system. SCDGs are thus an excellent candidate for analyzing the semantic behavior of potentially malicious programs, such as Mirai, that exploit the host system. Mining and classifying graphs is here achieved using the gSpan algorithm [41] that finds common sub-graphs. Experiments are conducted using gSpan for both signature extraction and classification. The results indicate that this semantic analysis is able to achieve better results that syntactic analysis. The rest of this section details the above.

5.1. Semantic Malware Classification

Similar semantic approaches have been considered before [14], [15], [30], [36] with system interaction behavior being used to classify malware. Typically these have some kind of graph structure similar to the SCDGs considered here, and some form of graph similarity used for classification. In particular multiple techniques are based on Control Flow Graphs as semantic signatures and use graph isomorphism to check matching [12], [22]; however, these techniques can be defeated by reordering non-dependent instructions [39], justifying the preference here for SCDGs instead.

In [15] the authors use graphs that relate vertices according to the number of shared parameters. For example, an edge label of 1 would indicate one shared parameter between the two system calls on the vertices. The gSpan algorithm is used to extract common sub-graphs over the whole graph set, and these are used as features for classification. Limiting to sub-graphs of at least two edges, the best result yielded 96.6% detection and 3.4% false positive.

In [30] the authors considered a similar conceptual approach. However, their SCDGs group similar system calls into categories and merge vertices from the same category, dependency edges are labeled only with the number of edges that existed prior to merging (and discard any parameter specific information). SCDGs are then compared with a δ-similarity metric and this is used for classification. Their results yielded 94.70% true positive and 13.10% false positive.

In [14] SCDGs are used as the behavioral representation of a program similar to in this work. However, their classification on graphs is done with social network properties (e.g. degree distribution, degree centrality, average distance, etc.) and various supervised learning algorithms. Results on three datasets ranged from Accuracy $90.19-99.97$ and AUC $0.87-1.00$.

Lastly in [36] a different behavioral approach was used; system interaction of a binary in a sandbox is observed and used to characterise behavior, i.e. usage of mutex's, changes to registry values, and induced error messages. These are first clustered to reduce the dimensionality, and then the resulting low-dimensional values are given to a random forest classifier.

5.2. Semantic Behavior Representation

Since system calls define the behavior by which a program interacts with its host system, SCDGs are an excellent semantic representation of a program. An SCDG is a graph whose vertices represent system calls and whose edges represent information flow between the system calls.

The motivation for SCDGs is that any actual action on the system (e.g. writing a file, sending a information through the network) requires a system call. Such a system call can be made directly by the program, or through a dynamic library function called by the program. For statically-linked binary programs such as Mirai, only the first option is available.

The behavior of a program can therefore by characterised by the system calls made during execution. However, a set of system calls alone can be easily altered (e.g. adding spurious calls, or changing the order of independent calls). Thus, to capture the required relations between the system calls, the data dependencies between calls are represented in an SCDG. Thus, a system call B is dependent upon the system call A if at least one argument of B is derived from a value returned or used by A. These dependencies are represented in the SCDG by directed edges annotated with a label noting which arguments of A are also arguments of B.

Definition 5.1 (SCDG)**.** Given a finite set of vertex labels $\mathcal{L}_\mathcal{V}$ and a finite set $\mathcal{L}_\mathcal{E}$ of edge labels, define a System Call Dependency Graph as a labeled directed graph $G = (\mathcal{V}, \mathcal{E}, \ell)$ where:

- $\mathcal{V}$ is a finite set of vertices,
- $\mathcal{E} \subset \mathcal{V} \times \mathcal{L}_\mathcal{E} \times \mathcal{V}$ is a set of labeled edges,
- $\ell : \mathcal{V} \to \mathcal{L}_\mathcal{V}$ is a function assigning a label $L_\mathcal{V} \in \mathcal{L}_\mathcal{V}$ to each vertex $V \in \mathcal{V}$.

In an SCDG a vertex represents at least one execution of the system call it is labeled with. Multiple vertices may exist with the same label, corresponding to different instances of that system call. An edge from A to B indicates that B uses information from A. Edges are labeled by pair of natural numbers and an arrow between them. An edge labeled by $i \to j$ indicates that information flows from argument i of A (0 being the return value) to the argument j of B. A small example of SCDGs is depicted in Figure 3.

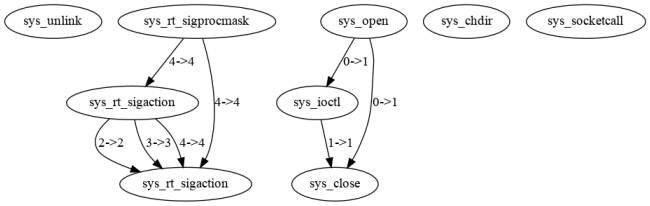

Figure 3. An example of an SCDG

5.3. Semantic Behavior Extraction

The extraction of an SCDG from a program is achieved by extracting execution *traces* of the program, generating the SCDG of each trace, and combining the SCDGs of these traces.

The trace of a program is extracted by symbolic execution using Angr [34]. Symbolic execution explores the various possible execution paths of the program by accumulating constraints along each possible execution path. Whenever a conditional is reached, two branches are created and only those with satisfiable constraints continue to be explored. This allows many traces to be generated that correspond to different possible executions of the program.

Note that since the program itself is not executed (only its possible traces are explored) the system calls are not actually executed. This has the advantage that in the case of a malicious program, the program cannot harm the host system. Angr provides the model of a few system calls, simulated using custom functions in python. In that case, the call can be symbolically executed. Otherwise, we rely on a database built from the Linux kernel source code in order to determine the number and type of arguments in the call. The call is simulated by using an unconstrained symbolic variable as return value.

Once a trace is generated, the dependencies between the system calls in the trace can be computed. This is a straightforward comparison of the arguments to the various system calls in the trace to detect the relations between them. The direction is determined by order of execution in the trace. For instance, the trace in Figure 4 yields the SCDG from 3.

```
1    syscall_stub_4_32 = sys_unlink(0)
2   syscall_stub_11_32 = sys_rt_sigprocmask(0, 2147417372, 0,          8)
3   syscall_stub_13_32 = sys_rt_sigaction(17, 2147415688, 2147415548, 8)
4  syscall_stub_129_32 = sys_rt_sigaction( 5, 2147415688, 2147415548, 8)
5                    3 = sys_open(134558446, 2, 0)
6  syscall_stub_236_32 = sys_ioctl(3, 2147768068, 2147417788)
7                    0 = sys_close(3)
8  syscall_stub_242_32 = sys_chdir(134558317)
9  syscall_stub_250_32 = sys_socketcall(1, 2147416080)
```

Figure 4. An example of a trace

At the end of the symbolic execution, all traces that are the prefix of another trace are discarded. The SCDGs of the remaining traces are generated, and their disjoint union is used to represent a single SCDG for the binary.

5.4. Graph Mining & Classification

This section overviews the extraction of a semantic signature from multiple SCDGS, and then how to use a semantic signature to classify a sample SCDG. Both exploit the gSpan algorithm [41] that given a set of graphs $\mathcal{G}$ finds common sub-graphs, i.e. given a set of graphs $\mathcal{G}$ and a percentage *support*, find all the common sub-graphs G' such that G' is a sub-graph of at least *support* of the different graphs $G \in \mathcal{G}$.

Once samples of SCDGs for Mirai have been collected, gSpan can be used to generate a semantic signature for Mirai as described in Algorithm 1. The signature is in practice the n largest sub-graphs that are common to at least *support* of the graphs in $\mathcal{G}$. This captures the n largest common components of behavior according to the SCDGs known to be Mirai.

Algorithm 1: Mirai Signature Generation with gSpan

Input : $\mathcal{G}$: set of Mirai SCDGs,
support: gSpan support parameter,
n: size of signature set
Output: $knowledge$: Semantic signature for mirai

1 *knowledge*= $\emptyset$
2 *CSG* $\leftarrow$ *gSpan*[$\mathcal{G}$, *support*] ▷ *support* common Mirai sub-graphs
3 *MaxCSG* $\leftarrow$ *sort_and_get(CSG, n)* ▷ Get the n largest common sub-graphs
4 **foreach** $G' \in MaxCSG$ **do**
5 *knowledge*$\leftarrow$ *knowledge*$\cup\{G'\}$

Observe that by finding common sub-graphs this ensures that common Mirai behavior is represented in the signature. This also eliminates SCDG components that are not common to many samples of Mirai, and so eliminates behavior that may be specific to a single instance.

Once a semantic signature for Mirai has been created, the classification of a new sample G is achieved (again

using gSpan) as shown in Algorithm 2. The inputs are the *knowledge* semantic signature generated from Algorithm 1, the sample G, and the *threshold* for similarity. The algorithm proceeds by considering each of the graphs in the semantic signature G'. gSpan is used to find the common sub-graphs of the sample G and the semantic signature graph G'. Then for each of the graphs G'' common to G' and G a similarity measure $\mathfrak{M}$ is computed. If this similarity measure is above the *threshold* then the similarity is sufficient to declare the sample as Mirai. Otherwise if no significant sub-graph commonality is found with any part of the Mirai semantic signature, then the sample is considered clean.

Algorithm 2: Classification with gSpan

Input : *knowledge*: the semantic signature from Algorithm 1,
G: graph of the sample to test,
threshold: the threshold for similarity to be malware
Output: Classification verdict

```
1 foreach (G') ∈ knowledge do
2     subsubG ← gSpan[{G',G}, 100%]
3     foreach G'' ∈ subsubG do
4         if 𝔐(G'', G') > threshold then
5             return Mirai
6 return clean
```

Here the similarity measure is simply to compare the number of edges of the two graphs to see how large the former is compared to the latter. That is

$$\mathfrak{M}(G'', G') \quad = \quad \frac{num\_edges(G'')}{num\_edges(G')} \, .$$

Observe that since G'' is a sub-graph of G' by definition, this is calculating how much of G' appears in G'' and in practice is included in the sample being tested. Hence, a similarity of 1 indicates that the semantic signature graph G' is entirely included in the sample.

5.5. Methodology & Results

This section describes the methodology for the experiments and the results achieved. A preliminary step was conducted to extract SCDGs and test the robustness of the SCDG extraction in Angr. After SCDG extraction, the methodology is divided into two parts, the first explored the parameters for the signature generation in Algorithm 1, and the second the threshold for the classification in Algorithm 2. Once these two parts have converged on parameters, additional experiments were conducted to validate the results.

The preliminary step of the methodology was to attempt to extract SCDGs from all the Mirai samples considered. In practice only 479 of the 516 Mirai samples were found to be able to generate traces using Angr. This yielded 37 Mirai samples where, due to lack of traces, no SCDGs could be extracted. These 37 failed SCDG extractions were not used in later experiments, since the failure of SCDG extraction in Angr cannot be determined to give either a positive or negative result. This also separates out failures due to Angr from failures of the later learning and classification experiments.

The preliminary step for the clean samples was to take 298 samples at random and extract their SCDGs. All 298 samples were able to have their SCDGs extracted successfully, so no further selection or exclusion was conducted on the clean samples.

Overall this yielded a sample set of 479 Mirai SCDGs and 298 clean SCDGs for use in the following experiments.

Experimenting for the parameters for the signature generation considered two parameters: the time allowed for gSpan graph mining, and the support size. The time parameter was a timeout that signaled gSpan to output graphs discovered so far, even if not completed. This was implemented to avoid pathological concerns, and also to explore incomplete or time-bounded usage of the signature generation. The support size was experimented upon to observe what percentage of the samples in the training set need to have a sub-graph for it to be significant as a semantic signature.

To explore the parameters in the first part, 20 test run sets were created, consisting of 383 Mirai samples for signature generation, and 96 Mirai samples and 149 clean samples for classification. The learning algorithm was then tested on these fixed 20 sets while varying the support from 0.4 to 0.8 in increments of 0.1, and the learning time from 500 to 2500 in increments of 1000.

To improve the efficiency of the experiments the exact similarity metric was computed for all sub-graphs as in line 4 of Algorithm 2, however this was stored to be compared with many different thresholds. The threshold were then tested in the range of 0.01 to 0.99 in increments of 0.01.

Table 3 reports the parameter combinations (learning time, support, and threshold) that achieved an $F_{0.5}$ score of greater than 99.50%. Observe that the best classification results are obtained for learning time of 2500s, support of 0.5, and threshold of $0.45 - 0.47$. Several other combinations of parameters also achieved comparably good results, indicating that the approach tends to be effective even with perturbations of the parameters.

Once the parameters for the signature generation and classification had both been determined, a larger number of experiments were run. Each experiment consisted of randomly selecting 383 of the Mirai samples to use in signature generation. The results were then used to classify the remaining Mirai samples and 178 of the clean samples. In total 540 instances of the above experiment were performed to reduce variance due to random sample selection and construction of training and classification sets. These results are summarized in Table 4.

Note that compared to Yara, the semantic approach improves the $F_{0.5}$-score by about 1%. The average learning time is 2503s, as set in the parameters. The average time for classifying the 274 graphs in the testing set is 1156s, which

TABLE 3. PARAMETERS COMBINATION YIELDING $F_{0.5}$ SCORE ABOVE 99.50% OVER 20 FIXED SELECTIONS OF LEARNING/TESTING GRAPHS.

Learning Time(s)	Support	Threshold	False Positive	False Negative	$F_{0.5}$ score
2500	50%	0.45 - 0.47	0.00%	0.99%	99.80%
2500	70%	0.45 - 0.47	0.00%	1.04%	99.79%
2500	60%	0.45 - 0.47	0.00%	1.04%	99.79%
1500	40%	0.4	0.00%	1.04%	99.79%
500	40%	0.4	0.00%	1.04%	99.79%
1500	80%	0.58 - 0.59	0.00%	1.09%	99.78%
1500	80%	0.6	0.00%	1.09%	99.78%
2500	80%	0.54 - 0.58	0.00%	1.09%	99.78%
500	40%	0.41	0.00%	1.09%	99.78%
1500	80%	0.54 - 0.56	0.03%	1.04%	99.75%
1500	80%	0.57	0.03%	1.09%	99.74%
1500	50%	0.45 - 0.47	0.00%	1.30%	99.74%
1500	60%	0.45 - 0.47	0.00%	1.30%	99.74%
1500	40%	0.38 - 0.39	0.07%	1.04%	99.70%
1500	80%	0.61 - 0.66	0.00%	1.46%	99.70%
1500	70%	0.45 - 0.47	0.00%	1.56%	99.68%
2500	70%	0.43 - 0.44	0.10%	1.04%	99.66%
500	80%	0.62 - 0.64	0.00%	1.82%	99.62%
2500	60%	0.43 - 0.44	0.13%	1.04%	99.62%
2500	50%	0.43 - 0.44	0.17%	0.99%	99.59%

TABLE 4. CONFUSION MATRIX AND CLASSIFICATION RESULTS OVER 540 EXPERIMENTS.

	Mirai samples	Clean samples
Detected Mirai	94.98 (98.94%)	0 (0%)
Detected Clean	1.02 (1.06%)	178 (100%)

Accuracy	Precision	$F_{0.5}$ score
99.63%	100.00%	99.78%

corresponds to about 4.2s per graph. Each experiment was run on a server with 120GB of RAM and 2 processors with 14 cores each allowing the parallel execution of 56 threads.

These results are restricted to the 479 graphs that we managed to extract out of our collection of 516 Mirai samples. If we account for these unextracted graphs as false negatives, this would increase the score of false negatives by about 7% (37 over 516). However, it is worth noting that all binaries undetected by YARA and VirusTotal were detected as Mirai by our approach when we managed to extract SCDGs from them. Reciprocally, all false negatives produced by our approach were detected as Mirai by either YARA or VirusTotal.

5.6. Limitations and Countermeasures

This section overview two main areas of limitations and also potential countermeasures to the approach used here. Lastly, a brief overview of tool based limitations and countermeasures is also considered.

The first area of limitations is the requirement of samples to begin the analysis from. Since the semantic signature required for semantic classification is synthesized from multiple samples, this requires multiple (accurately classified) samples to work from. Here the set of potential samples used to synthesize the semantic signature was 479 Mirai binaries. In any particular experiment however, only 383 were used.

The second area of limitations is computation and inherent to the underlying graph inclusion approach used, i.e. the gSpan algorithm. Here the approach limited the time spent in gSpan's graph mining. However, as evidenced in Section 5.5, larger time can be spent searching for larger (or even exact) common sub-graphs to use as signatures, since gSpan adopts a pattern-growth based (depth-first) approach where a frequent edge is extended recursively until all frequent sub-graphs are enumerated. Similarity between graphs within a specific support number and a certain timeout can be also be exploited. In practice this seems to give little further benefit since the results are already excellent.

A further computational limitation with gSpan is memory consumption since a large number of sub-graphs may be found during mining. For ideal semantic signature generation, isomorphic sub-graphs would be pruned from the search space. This improves the memory performance as well as solving problems with some pathological examples.

Countermeasures for the semantic approach here are more difficult to implement. The main way to achieve a countermeasure would be to significantly alter the structure of the SCDGs that are extracted. This requires changing the system calls used[2] or changing how the dependencies between the calls are created. (Aside, observe that simply adding more/spurious system calls does not remove the common sub-graph, only adds spurious graph structure that will be removed or ignored by the approach exploiting gSpan.)

A different path to considering both limitations and countermeasures is in the tools used. For example, a known limitation of Angr could be exploited to create a binary that the tools used here would fail to execute correctly (such as using packing and self modifying code). Another example would be to create a viable trace that yields a pathological graph for gSpan analysis (although this would need to be common to multiple samples or exploit particular dependencies to be effective). However, such counter-attacks are specific to the tools, and not the technique itself.

6. Combining the Syntactic and Semantic Approaches

The semantic approach in Section 5 obtains better results than the syntactic approach in Section 4. However, we do not recommend disposing of the syntactic approach, and instead recommend a combined syntactic-semantic approach. This combined approach uses each technique to cover for the shortcomings of the other.

Consider that the syntactic approach has negligible computational cost compared to the semantic approach, and in our experiments 0.00% false positives. This suggests that an unknown sample should first be analyzed syntactically: if

2. If synonymous calls can be found and these synonyms are not accounted for in the extraction and SCDG generation, as has been done in practice [30].

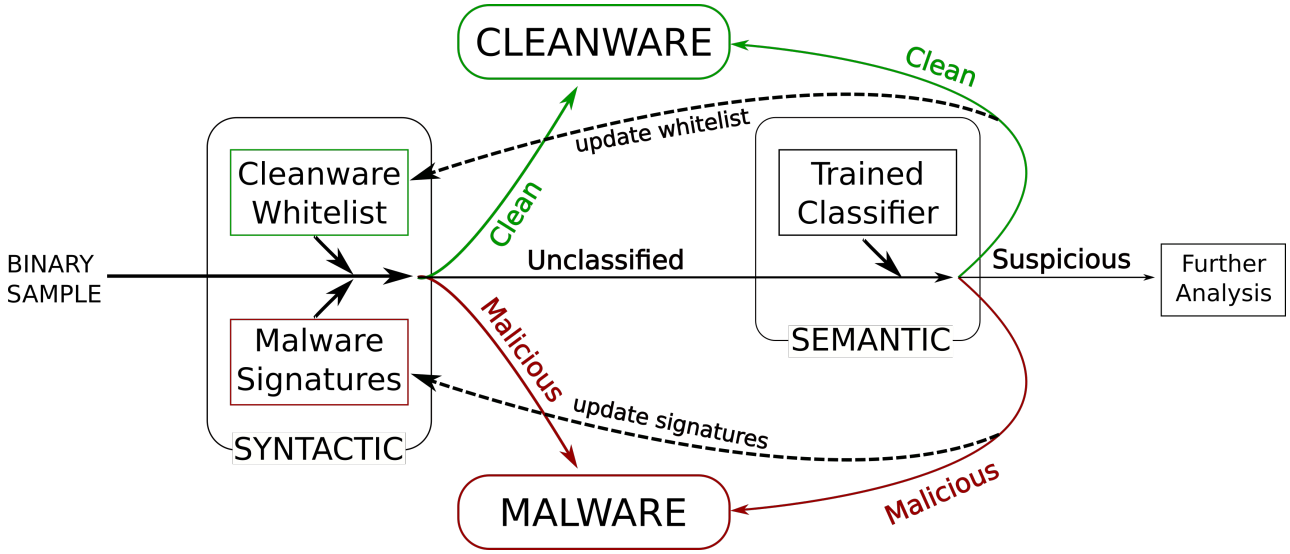

Figure 5. Malware detection using syntactic and semantic analysis combined. The binary sample is first analyzed syntactically, and classified if possible using malware signatures and a whitelist. If the sample is not classified by syntactic analysis, semantic analysis is used. If semantic analysis classifies the sample as clean or malware, it also updates the syntactic's analysis whitelist or malware signatures, respectively, to allow the sample to be detected syntactically in the future. If semantic analysis finds the file suspicious but not enough to classify it as malware, it is produced for further analysis (manual or automated).

this is sufficient to classify the sample, there is no reason to spend ulterior computational power to extract the sample's SCDG and run the semantic approach. In particular, the low computational cost of the syntactic approach is the reason why time-bound devices like firewalls limit themselves to syntactic analysis.

If the syntactic approach fails to classify a sample then the semantic approach should be executed. If a sample is classified as malicious by the semantic approach, it is convenient to update the syntactic signature to also recognize that sample in the future. This ensures that the cost of semantic analysis will not have to be paid for the sample if it is analyzed again. In the firewall scenario, samples that fail to be analyzed syntactically can be sent to a different server for semantic analysis. Information about the malicious samples that avoided syntactic analysis but were detected by semantic analysis can be added to the syntactic signatures to detect them at the syntactic analysis level in the future.

Observe that although the experiments for semantic analysis yielded no false positives, typically a small number of false positives are found with semantic techniques. This can be easily solved in many cases by whitelisting of known clean binaries, such as system files. The whitelist then forms part of the syntactic analysis and prevents expensive semantic analysis of known clean samples.

This combined approach is summarized in Figure 5.

The results are the experiments (in Sections 4 & 5) are here combined to estimate the efficiency of such a combined approach. Since neither of the approaches produced false positive, the focus here is on the false negatives. The first pass of YARA on the sample set of 516 Mirai samples correctly classifies 497 of them as Mirai. Among the remaining 19 samples, SCDG extraction failed on 2 of them

TABLE 5. CONFUSION MATRIX AND CLASSIFICATION RESULTS OF THE COMBINED APPROACH.

	Mirai samples	Clean samples
Detected Mirai	510 (98.84%)	0 (0%)
Detected Clean	6 (1.16%)	6438 (100%)

Accuracy	Precision	$F_{0.5}$ score
99.91%	100.00%	99.77%

that were thus not detected by semantic analysis, and 4 had their SCDG extracted but classified as clean. The combined approach thus correctly classifies all but 6 samples. The corresponding results are given in Table 5. Note that even if the $F_{0.5}$-score is below the one achieved in Section 5, the results presented here take into account the SCDGs that failed to be extracted in Section 5. Therefore, Table 5 demonstrates that complementing the YARA approach with the semantic approach here increases the $F_{0.5}$-score from 98.69% (YARA only) to 99.77% (combined approach), while keeping the computational cost minimal (since the relatively expensive semantic approach is executed only on 19 samples).

Finally, note that the 4 Mirai samples that were extracted but not detected by semantic analysis were actually detected as Olyx by Yara. Each of these four samples has a score around 21% (they contain 21% of the edges of a subgraph characteristic of Mirai), which is below the threshold of 45%, and less than some clean samples. Since these 4 samples have been classified as Mirai by the majority of the engines run by VirusTotal that detected them, they are considered to be Mirai here. However, further manual analysis of these samples to determine their classification and similarity is ongoing work.

7. Conclusions

This paper explores malware detection techniques to detect samples of the Mirai botnet trojan. Two techniques are evaluated: syntactic analysis with string-based signatures and detection based on the YARA tool; and semantic analysis with system-call-based signatures and detection based on machine learning with the gSpan common sub-graph algorithm. Both techniques are tested on a set of more than 500 unique Mirai samples captured using a honeypot, and on a set of cleanware. While both techniques have zero false positives on the sample set, semantic analysis is found to be more effective at detecting Mirai samples than syntactic analysis, with semantic analysis having an $F_{0.5}$-score of 99.78% of the samples against the $F_{0.5}$-score of 98.69% of syntactic analysis.

Due to the lower cost of syntactic analysis compared to semantic analysis, we recommend to use both techniques in increasing cost order in a combined approach, using information from the semantic technique to improve detection of the syntactic technique and reduce false positives for both approaches. In the network detection scenario, this means running only syntactic detection in the routers/firewalls and submitting suspicious samples to a semantic analysis system hosted on a server with greater computational power. The syntactic detection approach can then be reinforced by updating its signatures to be able to detect the samples that escape it and are detected by the semantic detection approach.

References

[1] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan. N-gram-based detection of new malicious code. In *Proceedings of the 28th Annual International Computer Software and Applications Conference - Workshops and Fast Abstracts - Volume 02*, COMPSAC '04, pages 41–42, Washington, DC, USA, 2004. IEEE Computer Society.

[2] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto. Novel feature extraction, selection and fusion for effective malware family classification. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 183–194, New York, NY, USA, 2016. ACM.

[3] V. M. Álvarez. *YARA*. VirusTotal, August 2016. http://virustotal.github.io/yara/.

[4] K. Beaumont. *Shadows Kill: Mirai DDoS botnet testing large scale attacks, sending threatening messages about UK and attacking researchers*. Medium, November 2016. https://medium.com/@networksecurity/shadows-kill-mirai-ddos-botnet-testing-large-scale-attacks-sending-threatening-messages-about-6a61553d1c7.

[5] D. Bekerman. *New Mirai Variant Launches 54 Hour DDoS Attack against US College*. Imperva Incapsula, March 2017. https://www.incapsula.com/blog/new-mirai-variant-ddos-us-college.html.

[6] D. Bilar. Opcodes as predictor for malware. *Int. J. Electron. Secur. Digit. Forensic*, 1(2):156–168, Jan. 2007.

[7] S. Bühlmann. *Introduction to YARA Rule Generator*. JoeSecurity, February 2015. http://blog.joesecurity.org/2015/02/introduction-yara-rule-generator.html.

[8] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association.

[9] C. Clark. *YaraGenerator*. XenoSec, August 2013. https://yaragenerator.com/.

[10] J. Costello, A. Nixon, B. Hein, R. Tokazowski, and Z. Wikholm. *New Mirai Variant Leaves 5 Million Devices Worldwide Vulnerable High Concentration in Germany, UK and Brazil*. Flashpoint, November 2016. https://www.flashpoint-intel.com/blog/cybercrime/new-mirai-variant-involved-latest-deutsche-telekom-outage/.

[11] C. Douligeris and A. Mitrokotsa. Ddos attacks and defense mechanisms: Classification and state-of-the-art. *Comput. Netw.*, 44(5):643–666, Apr. 2004.

[12] H. Flake. Structural comparison of executable objects. In U. Flegel and M. Meier, editors, *Detection of Intrusions and Malware & Vulnerability Assessment, GI SIG SIDAR Workshop, DIMVA 2004, Dortmund, Germany, July 6.7, 2004, Proceedings*, volume 46 of *LNI*, pages 161–173. GI, 2004.

[13] B. Herzberg, D. Bekerman, and I. Zeifman. *Breaking Down Mirai: An IoT DDoS Botnet Analysis*. Imperva Incapsula, October 2016. https://www.incapsula.com/blog/malware-analysis-mirai-ddos-botnet.html.

[14] J.-w. Jang, J. Woo, A. Mohaisen, J. Yun, and H. K. Kim. Malnetminer: Malware classification approach based on social network analysis of system call graph. *Mathematical Problems in Engineering*, 2015, 2015.

[15] F. Karbalaie, A. Sami, and M. Ahmadi. Semantic malware detection by deploying graph mining. *International Journal of Computer Science Issues*, 9(1):373–379, 2012.

[16] M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1(1):13–23, Nov 2005.

[17] O. Klaba. *Last days, we got lot of huge DDoS. Here, the list of "bigger that 100Gbps" only. You can see the simultaneous DDoS are close to 1Tbps !* OVH [twitter user @olesovhcom], September 2016. https://twitter.com/olesovhcom/status/778830571677978624.

[18] J. Z. Kolter and M. A. Maloof. Learning to detect malicious executables in the wild. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 470–478, New York, NY, USA, 2004. ACM.

[19] B. Krebs. *Did the Mirai Botnet Really Take Liberia Offline?* KrebsOnSecurity, November 2016. https://krebsonsecurity.com/2016/11/did-the-mirai-botnet-really-take-liberia-offline/.

[20] B. Krebs. *KrebsOnSecurity Hit With Record DDoS*. KrebsOnSecurity, September 2016. https://krebsonsecurity.com/2016/09/krebsonsecurity-hit-with-record-ddos/.

[21] B. Krebs. *Source Code for IoT Botnet 'Mirai' Released*. KrebsOnSecurity, October 2016. https://krebsonsecurity.com/2016/10/source-code-for-iot-botnet-mirai-released/.

[22] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, RAID'05, pages 207–226, Berlin, Heidelberg, 2006. Springer-Verlag.

[23] J. Liu, Y. Xiao, K. Ghaboosi, H. Deng, and J. Zhang. Botnet: Classification, attacks, detection, tracing, and preventive measures. *EURASIP Journal on Wireless Communications and Networking*, 2009(1):692654, Sep 2009.

[24] MalwareMustDie! *MMD-0056-2016 - Linux/Mirai, how an old ELF malcode is recycled..*, August 2016. http://blog.malwaremustdie.org/2016/08/mmd-0056-2016-linuxmirai-just.html.

[25] MalwareTech. *Mapping Mirai: A Botnet Case Study*, October 2016. https://www.malwaretech.com/2016/10/mapping-mirai-a-botnet-case-study.html.

[26] G. Marsaglia. Xorshift rngs. *Journal of Statistical Software, Articles*, 8(14):1–6, 2003.

[27] E. Menahem, A. Shabtai, L. Rokach, and Y. Elovici. Improving malware detection by applying multi-inducer ensemble. *Computational Statistics & Data Analysis*, 53(4):1483 – 1494, 2009.

[28] J. Mirkovic, S. Dietrich, D. Dittrich, and P. Reiher. *Internet Denial of Service: Attack and Defense Mechanisms (Radia Perlman Computer Networking and Security)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[29] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430, Dec 2007.

[30] S. D. Nikolopoulos and I. Polenakis. A graph-based model for malicious code detection exploiting dependencies of system-call groups. In B. Rachev and A. Smrikarov, editors, *Proceedings of the 16th International Conference on Computer Systems and Technologies, CompSysTech, Dublin, Ireland, June 25 - 26, 2015*, pages 228–235. ACM, 2015.

[31] A. Nixon, J. Costello, and Z. Wikholm. *An After-Action Analysis of the Mirai Botnet Attacks on Dyn*. Flashpoint, October 2016. https://www.flashpoint-intel.com/blog/cybercrime/action-analysis-mirai-botnet-attacks-dyn/.

[32] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, SP '01, pages 38–, Washington, DC, USA, 2001. IEEE Computer Society.

[33] A. Shabtai, R. Moskovitch, Y. Elovici, and C. Glezer. Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. *Inf. Secur. Tech. Rep.*, 14(1):16–29, Feb. 2009.

[34] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[35] M. Siddiqui, M. C. Wang, and J. Lee. Data mining methods for malware detection using instruction sequences. In *Proceedings of the 26th IASTED International Conference on Artificial Intelligence and Applications*, AIA '08, pages 358–363, Anaheim, CA, USA, 2008. ACTA Press.

[36] J. Stiborek, T. Pevnỳ, and M. Rehák. Multiple instance learning for malware classification. *arXiv preprint arXiv:1705.02268*, 2017.

[37] S. J. Stolfo, K. Wang, and W.-J. Li. *Towards Stealthy Malware Detection*, pages 231–249. Springer US, Boston, MA, 2007.

[38] A. H. Sung, J. Xu, P. Chavez, and S. Mukkamala. Static analyzer of vicious executables (SAVE). In *20th Annual Computer Security Applications Conference*, pages 326–334, Dec 2004.

[39] G. R. Thompson and L. A. Flynn. Polymorphic malware detection and identification via context-free grammar homomorphism. *Bell Labs Technical Journal*, 12(3):139–147, Fall 2007.

[40] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy*, pages 674–691, May 2015.

[41] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM '02, pages 721–, Washington, DC, USA, 2002. IEEE Computer Society.

Appendix

This section presents details of Mirai that have not been included in the main paper. Table 6 presents the list of subnet addresses that are avoided by the Mirai bot when generating addresses for possible infection. Table 7 presents the list of default username/password pairs used by the Mirai bot for infection.

```
127.0.0.0/8 Loopback
0.0.0.0/8 Invalid address space
3.0.0.0/8 General Electric Company
15.0.0.0/7 Hewlett-Packard Company
56.0.0.0/8 US Postal Service
10.0.0.0/8 Internal network
192.168.0.0/16 Internal network
172.16.0.0/14 Internal network
100.64.0.0/10 IANA NAT reserved
169.254.0.0/16 IANA NAT reserved
198.18.0.0/15 IANA Special use
224-255.0.0.0/8 Multicast
6.0.0.0/8 Department of Defense
7.0.0.0/8 Department of Defense
11.0.0.0/8 Department of Defense
21.0.0.0/8 Department of Defense
22.0.0.0/8 Department of Defense
26.0.0.0/8 Department of Defense
28.0.0.0/8 Department of Defense
29.0.0.0/8 Department of Defense
30.0.0.0/8 Department of Defense
33.0.0.0/8 Department of Defense
55.0.0.0/8 Department of Defense
214.0.0.0/8 Department of Defense
215.0.0.0/8 Department of Defense
```

TABLE 6. SUBNET ADDRESSES SPECIFICALLY AVOIDED BY THE MIRAI BOT.

```
666666 666666
888888 888888
admin
admin 1111
admin 1111111
admin 1234
admin 12345
admin 123456
admin 54321
admin 7ujMko0admin
admin admin
admin admin1234
admin meinsm
admin pass
admin password
admin smcadmin
admin1 password
administrator 1234
Administrator admin
guest 12345
guest guest
mother fucker
root
root 00000000
root 1111
root 1234
root 12345
root 123456
root 54321
root 666666
root 7ujMko0admin
root 7ujMko0vizxv
root 888888
root admin
root anko
root default
root dreambox
root hi3518
root ikwb
root juantech
root jvbzd
root klv123
root klv1234
root pass
root password
root realtek
root root
root system
root user
root vizxv
root xc3511
root xmhdipc
root zlxx.
root Zte521
service service
supervisor supervisor
support support
tech tech
ubnt ubnt
user user
```

TABLE 7. DEFAULT USERNAME/PASSWORD PAIRS LIST USED BY THE MIRAI BOT. NOTE THAT THE BOT CONTAINS 62 USERNAME/PASSWORD PAIRS AS ENCODED STRINGS, BUT TWO PAIRS - ADMIN/1234 AND GUEST/1234 - ARE PRESENT TWICE EACH.