Virus Bulletin :: VB2018 paper: Tracking Mirai variants

Archived: 2026-04-05 15:09:53 UTC

Ya Liu & Hui Wang

Qihoo 360 Technology, China

Copyright © 2018 Virus Bulletin

Abstract

Mirai, the infamous DDoS botnet family known for its great destructive power, was made open source soon after being found by *MalwareMustDie* in August 2016, which led to a proliferation of Mirai variant botnets. Since then, the authors have continuously updated their code (e.g. adding new types of exploits and attack methods), which has added to the difficulty of detecting and mitigating Mirai-related threats. To solve that problem we present a set of Mirai variant classification and tracking schemes developed during the process of analysing over 32,000 Mirai samples. In this paper we will introduce:

1. How to extract data including configurations, supported attack methods, and dictionaries of usernames and passwords from samples.
2. How to use the extracted data to classify and track Mirai variants.

To demonstrate the effectiveness of our solutions, popular Mirai branches are investigated under the proposed schemes.

1. Introduction

Mirai became well known in Autumn 2016 for overwhelming several high-profile targets including *Krebs on Security*, *OVH* and *Dyn* through DDoS attacks. Mirai overtook previous *Linux* DDoS botnet families (e.g. Gafgyt, Tsunami) in its capacity to infect hundreds of thousands of IoT devices in a short period of time, and to provide versatile attack method options. The Mirai source code was released soon after having been found by *MalwareMustDie*. Inspired by the success of Mirai and the released source code, other bot masters/underground groups soon began to establish their own versions of Mirai botnets, which has caused a proliferation of IoT botnets over the past 1.5 years. While some of the new botnets only borrowed ideas or code from Mirai (e.g. Hajime [1], Reaper [2]), most of them are exact Mirai descendants [3, 4, 5]. In the post-Mirai era, it should be routine work for security researchers to fight Mirai-like threats.

Currently, it is usual for Mirai variants to be classified with their branch names, which come from the command line '/bin/busybox <*branch* >' found in the Mirai sample. While the default name is 'MIRAI', in later variants the 'branch' is usually replaced with an author-chosen name (e.g. MASUTA, SATORI, SORA, as shown in Table 1). However, we feel that such a classification scheme is too coarse-grained, and it cannot reveal the variances across variants.

Branch name	Sample count
JOSHO	1,444
OWARI	702
MASUTA	438
Cult	434
SORA	400
daddyl33t	343

MIORI	244
WICKED	128
dwickedgod	125
EXTENDO	100

Table 1: Top 10 new Mirai branches in the past 1.5 years based on their sample count.

Our Mirai tracking work started soon after the malware was first blogged about by *MalwareMustDie* in August 2016 [6]. Between then and May 2018, over 32,000 Mirai samples were collected, from which dozens of variants and 1,000+ C&Cs have been detected. According to our experience, automatic schemes to classify and track the proliferated variants must be able to do the following:

1. Extract the C&C information.
2. Figure out the supported attack methods, since Mirai is mainly DDoS attack purposed.
3. Provide clues for correlating variants and C&Cs, and if possible, help to investigate the actors behind the attacks.

In this paper, we will introduce our Mirai variant classification and tracking schemes which mainly make use of the data relating to configurations, supported attack methods and credential dictionaries. The remainder of this paper is organized as follows: in Section 2, we introduce how to extract data including configurations, supported attack methods and attack method fingerprints automatically from samples; in Section 3, we introduce a set of schemes including variant clustering and classification using the extracted data; in Section 4 we use our proposed schemes to investigate some typical Mirai branches.

To summarize, the contributions of this paper are as follows:

- We demonstrate solutions for automatically extracting configurations, supported attack methods and credential dictionaries from Mirai samples.
- We propose a fingerprint technique that can be used to recognize Mirai attack methods with the information extracted from binary code without the need for reverse engineering work.
- We summarize the Mirai variants and introduce a set of classification schemes based on the extracted data.
- We investigate popular Mirai branches with our proposed schemes.

Since it's common for Mirai botnet authors to compile the same code into binaries for different processors (e.g. x86, x64, ARM, MIPS, SPARC, PowerPC), for reasons of simplicity and efficiency, we chose to consider only the subset of samples for x86 and ARM in this paper. We believe these two kinds of samples are sufficient to study the Mirai variants due to the redundancy introduced by the 'one-source-to-multiple-processors' style of compilation. On the other hand, this paper mainly focuses on Mirai variants with DDoS attack methods, and pays little attention to non-DDoS ones (e.g. Sartori.miner).

The SHA256 hashes for the samples discussed in this paper are given in Appendix A.

2. Data extraction

It's thought that malware variant classification and tracking is a classical, yet difficult problem. There is no industrial standard on that. The de facto method is to make use of sample information including special code snippets, binary byte sequences or strings, calling function graphs, and size, etc. In this paper, we simplify the problem by limiting the used data to the following four kinds:

1. Plaintext configurations, together with the encryption algorithm and key.
2. The C&C domain/IP and port. While originally stored in the configuration database, this information is now usually hard coded in a function named resolve_cnc_addr().
3. The supported attack methods and their corresponding command codes.
4. The dictionary of username/password pairs used in the scan module, if it exists.

All the data is extracted in an automated manner. Since the data is stored in a distributed manner in the samples (rather than all being in the same place), it's difficult to extract all the required data with a single solution. Meanwhile, due to the 'one-source-to-multiple-processors' code compilation, solutions must be able to deal with different processor architectures. Our final solution is composed of four separate analyser programs, each of which includes a static and a dynamic analysis part. The static analysis is done with IDAPython to make use of *IDA*'s multiple processor supporting feature, while the dynamic analysis is based on a proprietary lightweight emulation framework which is designed to emulate binary code snippets of interest (e.g. an instruction block). The open-source emulator *Unicorn* [7] is used as the core engine to support common processor architectures including x86/x64/ARM/MIPS/PowerPC/SPARC. Both ELF and PE formats are supported.

2.1 Configurations

In Mirai, a self-defined database is designed to store most of the running parameters which we call configurations. The database has the following characteristics:

1. Each item is uniquely indexed with a number which is usually less than 256.
2. Configurations are encrypted.
3. Configuration is referenced in a pattern of 'decrypt->retrieve->re-encrypt'.

A completely extracted configuration database can be seen in Figure 1.

```
CNC=indiatechsupport.club, port=1337, cnc_hardcoded_in_rsca=true, arch=arml, idx_port=1, key=0x54, alg=1
[0x01]: "\x059", size=2
[0x02]: "\x07\xbe", size=2
[0x03]: "DaddyL33T Infected Your Shit\x00", size=29
[0x04]: "shell\x00", size=6
[0x05]: "enable\x00", size=7
[0x06]: "system\x00", size=7
[0x07]: "sh\x00", size=3
[0x08]: "/bin/busybox JOSHO\x00", size=19
[0x09]: "JOSHO: applet not found\x00", size=24
[0x0a]: "ncorrect\x00", size=9
[0x0b]: "/bin/busybox ps\x00", size=16
[0x0c]: "/bin/busybox kill -9 \x00", size=22
[0x0d]: "/proc/\x00", size=7
[0x0e]: "/exe\x00", size=5
[0x0f]: "/fd\x00", size=4
[0x10]: "/maps\x00", size=6
[0x11]: "/proc/net/tcp\x00", size=14
[0x1d]: "/status\x00", size=8
[0x1e]: ".anime\x00", size=7
[0x12]: "/proc/net/route\x00", size=16
[0x13]: "assword\x00", size=8
[0x14]: "TSource Engine Query\x00", size=21
[0x15]: "/etc/resolv.conf\x00", size=17
[0x16]: "nameserver \x00", size=12
[0x17]: "/dev/watchdog\x00", size=14
[0x18]: "/dev/misc/watchdog\x00", size=19
[0x19]: "pbbf~cu\x11", size=8
[0x1a]: "ogin\x00", size=5
[0x1b]: "enter\x00", size=6
[0x1c]: "1gba4cdom53nhp12ei0kfj\x00", size=23
```

Figure 1: A successfully extracted configuration database (md5 = 0ae272306d313c6abf1433b85e0a2352).

The first line holds the summary information, while the left lines correspond to individual configuration items. In this example the C&C port has an index number of 1, as indicated by 'idx_port=1' in the first line, which points to the item '[0x01]: "\x059", size=2'. The C&C port of 0x539 (a.k.a. 1337) is calculated from the item content of '\x059'.

Basically, configuration extraction can be divided into two steps: (1) extracting the indexes and cipher texts of all items; (2) deciphering them with a self-implemented decryption program. Our analysis of the source shows that the database is initialized in a function called table_init(), as shown in Figure 2.

```
void table_init(void)
{
    add_entry(TABLE_CNC_DOMAIN, "\x41\x4C\x41\x0C\x41\x4A\x43\x4C\x45\x47\x4F\x47\x0C\x41\x4D\x4F\x22", 30); // cnc.changeme.com
    add_entry(TABLE_CNC_PORT, "\x22\x35", 2);   // 23

    add_entry(TABLE_SCAN_CB_DOMAIN, "\x50\x47\x52\x4D\x50\x56\x0C\x41\x4A\x43\x4C\x45\x47\x4F\x47\x0C\x41\x4D\x4F\x22", 29); // report.changeme.com
    add_entry(TABLE_SCAN_CB_PORT, "\x99\xC7", 2);         // 48101

    add_entry(TABLE_EXEC_SUCCESS, "\x4E\x4B\x51\x56\x47\x4C\x4B\x4C\x45\x02\x56\x57\x4C\x12\x22", 15);

    // safe string https://youtu.be/dQw4w9WgXcQ
    add_entry(TABLE_KILLER_SAFE, "\x4A\x56\x56\x52\x51\x18\x0D\x0D\x5B\x4D\x57\x56\x57\x0C\x40\x47\x0D\x46\x73\x55\x16\x55\x1B\x75\x45\x7A\x41\x73\x22", 29);
    add_entry(TABLE_KILLER_PROC, "\x0D\x52\x50\x4D\x41\x0D\x22", 7);
    add_entry(TABLE_KILLER_EXE, "\x0D\x47\x5A\x47\x22", 5);
    add_entry(TABLE_KILLER_DELETED, "\x02\x0A\x46\x47\x4E\x47\x56\x47\x46\x0B\x22", 11);
    add_entry(TABLE_KILLER_FD, "\x0D\x44\x46\x22", 4);
    add_entry(TABLE_KILLER_ANIME, "\x0C\x43\x4C\x4B\x4F\x47\x22", 7);
    add_entry(TABLE_KILLER_STATUS, "\x0D\x51\x56\x43\x56\x57\x51\x22", 8);
    add_entry(TABLE_MEM_QBOT, "\x70\x67\x72\x6D\x70\x76\x02\x07\x51\x18\x07\x51\x22", 13);
    add_entry(TABLE_MEM_QBOT2, "\x6A\x76\x76\x72\x64\x6E\x6D\x6D\x66\x22", 10);
    add_entry(TABLE_MEM_QBOT3, "\x6E\x6D\x6E\x6C\x6D\x65\x76\x64\x6D\x22", 10);
    add_entry(TABLE_MEM_UPX, "\x7E\x5A\x17\x1A\x7E\x5A\x16\x66\x7E\x5A\x16\x67\x7E\x5A\x16\x67\x7E\x5A\x16\x11\x7E\x5A\x17\x12\x7E\x5A\x16\x14\x7E\x5A\x10\x10\x22", 3
    add_entry(TABLE_MEM_ZOLLARD, "\x58\x4D\x4E\x4E\x43\x50\x46\x22", 8);
    add_entry(TABLE_MEM_REMAITEN, "\x65\x67\x76\x6E\x6D\x61\x63\x6E\x6B\x72\x22", 11);
```

Figure 2: The table_init() function in source.

In table_init(), it's the function named 'add_entry()' that is called repeatedly to install the individual items. In each call to add_entry(), three parameters are passed which individually represent: item index, ciphertext address, and size. A new memory block will be allocated inside add_entry() to store the cipher text, then it will be saved to a slot allocated from a global structure named 'table'. The slot position is determined with the following formula:

```
item_addr=table_addr +item_id*tbl_item_size
```

Under 32-bit CPU architecture, the tbl_item_size always holds a value of 8, while its value is 16 for 64-bit CPU architectures.

In the original versions, both the C&C domain/IP and port are stored in the configuration database, which makes it possible to recover all necessary information simply by analysing the table_init() function. However, in later variants, C&C

domains/IPs are usually hard coded in a function called

resolve_cnc_addr(), as shown in Figure 1. Furthermore, due to compiler inline optimization and the fact that the first item is not always indexed from a fixed number (e.g. 0), configuration indexing information is missing in table_init().

Fortunately, since both C&C and port are referenced in resolve_cnc_addr(), the missing indexing information, together with the hard-coded C&C, can be calculated heuristically with the analysis of resolve_cnc_addr(). Therefore our configuration extraction solution comes as two analyser programs: one for table_init() and the other for resolve_cnc_addr().

2.1.1 table_init() analyser

The table_init() function is called after the bot starts running. As shown in Figure 2, the item installation work is done by repeatedly calling add_entry(). In the case of inline optimization, calls to add_entry() are optimized into those of malloc()/util_memcpy().

The 'decrypt->retrieve->re-encrypt' style of item reference is done separately with the table_unlock_val()/table_retrieve_val()/table_lock_val() functions, with decryption/encryption implemented in table_unlock_val()/table_lock_val(). The encryption algorithm is summarized as follows:

1. Decryption and encryption share the same single-byte XOR'ing algorithm.
2. Key size is 4. The target byte is XOR'ed with each key byte in turn to get the final ciphertext/plaintext byte.

Given the associative law of XOR operation, the four-byte key can equivalently be mapped to a one-byte key, which greatly reduces the key space from 2^32 to 2^8, thus making it feasible to search the key with brute force enumeration.

Candidate table_init() functions are found by the static analysis script which goes through all binary functions in samples and picks out those with the following characteristics:

- Repeatedly calling add_entry() or malloc()/util_memcpy() in the case of inline optimization.
- Composed of one very large instruction block because no branches are introduced by any switch/loop instructions.
- Referencing dozens of global variables which point to ciphertext memory.

Dynamic analysis of the candidate table_init() functions, together with false-positive removals, is done in an emulation program based on our lightweight emulation framework. The cases of non-inline and inline optimization are considered separately. The key points of emulating a non-inline version of table_init() are as follows:

- NOP'ing all the CALL instructions and marking them as breakpoints.
- Emulating in single-step mode.
- In the breakpoint handler, saving arguments 1–3 separately as id, ciphertext address, and size if argument 2 points to a valid data area.

When dynamically analysing the inline optimized code, a trick is used which is inspired by the fact that the return value of malloc() would be used as the first argument of the next call (a.k.a. util_memcpy()). Every time a CALL breakpoint is handled, the return value (e.g. EAX in the case of *Intel* x86 CPU) will be set to a magic value called MAGIC-RET. By doing that, a call with the first argument equal to MAGIC-RET would be indicative of util_memcpy, thus triggering the saving of arguments 2 and 3. Similarly, the operation of saving an item to the table can also be traced by checking whether the currently inspected MEM-WRITE operation has a source operand of MAGIC-RET. If it does, the slot address will be saved for late index calculation with the following formula:

```
item_id=(item_addr-table_addr)/tbl_item_size
```

The table_addr is inferred heuristically from the analysis of resolve_cnc_addr() introduced in Section 2.1.2.

The key points of emulating an inline optimized table_init() are as follows:

1. NOP'ing all the CALL instructions and marking them as breakpoints.
2. Hooking all MEM-WRITE instructions.
3. Emulating in single-step mode.
4. In the CALL breakpoint handler, saving arguments 2 and 3 as ciphertext address and size if the first argument is MAGIC-RET, and setting the return value as MAGIC-RET.
5. In the MEM-WRITE handler, saving the destination memory address as item_addr if the source operand is MAGIC-RET.

After emulation finishes, the ciphertext configurations are read for decryption. Every key in 2^8 is tested until any meaningful plain configurations are found. The decrypted configurations will be cached for later synthesizing after resolve_cnc_addr() is analysed.

2.1.2 resolve_cnc_addr() analyser

The resolve_cnc_addr() function, called in a SIGTRAP signal handler installed in main(), was originally responsible for retrieving the C&C and port from the configuration database with their index numbers. However, more and more Mirai variants have begun to hard code their C&Cs in resolve_cnc_addr(), as illustrated in Figure 3, which mandates the analysis of resolve_cnc_addr() to retrieve the right C&C value and port index number. The result will be used for inference of configuration indexing, as mentioned above.

```
resolve_cnc_addr proc near

var_1C= dword ptr -1Ch

sub     esp, 18h
push    2
call    table_unlock_val
pop     eax
pop     edx
push    0
mov     dword ptr ds:srv_addr+4, 4C4D35Fh
push    2
call    table_retrieve_val
mov     ax, [eax]
mov     [esp+1Ch+var_1C], 2
mov     word ptr ds:srv_addr+2, ax
call    table_lock_val
add     esp, 1Ch
retn
resolve_cnc_addr endp
```

Figure 3: C&C hard coded in resolve_cnc_addr() (md5 = 333d98e27cc885624f073e59fc40dfed).

The resolve_cnc_addr() function has the following characteristics:

1. It writes at least two global variables (srv_addr.sin_addr and srv_addr.sin_port).
2. It calls at least three different functions.

Candidate resolve_cnc_addr() functions are found with an IDAPython script in the following ways:

1. Heuristically finding the main() function.
2. Finding all callback functions in main() as candidate signal handlers.
3. Finding callback functions in each candidate signal handler found in (2) as candidate resolve_cnc_addr() functions.
4. Filtering out those that don't have reserve_cnc_addr() characteristics.
5. Extracting the hard-coded C&C if it exists.

The key steps of emulating resolve_cnc_addr() are as follows:

1. NOP'ing all the CALL instructions and marking them as breakpoints.
2. Hooking all MEM-WRITE instructions.
3. Emulating in single-step mode.
4. In the CALL breakpoint handler, saving a pair of {arg1, 'CALL'} to an operation list OPS, and setting the return value (e.g. EAX in the case of *Intel* x86) as a magic value MAGIC-RET.
5. In the MEM-WRITE handler, saving a pair of {write-bytes, 'WRITE'} to OPS.

After emulation stops, the C&C (if not hard coded) and port index numbers are calculated in the following manner:

1. The OPS list is iterated and the unique arg1 values saved in {arg1, 'CALL'} pairs are counted until a {write-byes, 'WRITE'} pair is encountered.
2. The most frequent arg1 value can be thought of as an index number. If write-bytes in {write-byes, 'WRITE'} is equal to 4, the obtained number will be used as the C&C index; if write-bytes is equal to 2, it is used as the C&C port

index.
3. The counting results are cleared for the next round of counting.
4. Steps 1 to 3 are repeated until OPS is iterated over.

The indexes that are found are used for C&C retrieval and for inferring the indexing number in the case of inline optimized table_init(). The heuristic inference is based on the facts that: (1) although not fixed, the smallest index number must be greater than 0 while usually less than 5; (2) the C&C port item always has a size of 2. The detailed process is illustrated as follows:

1. Getting the first item's address by sorting the values of item_addr obtained in Section 2.1.1, and taking it as first_item_addr.
2. Assuming the value of first_item_index is 1.
3. Calculating the table address with the formula: table_addr (=first_item_addr- first_item_index*item_size).
4. Calculating all other item indexes based on the table_addr obtained in step 3 and the item_addr obtained in Section 2.1.1.
5. Retrieving the C&C and port with indexes obtained from resolve_cnc_addr(). If the retrieved values make sense, the process stops here.
6. Increasing the first_item_index by 1, and going back to step 3 if the first_item_index is less than 5. Otherwise the process stops.

An example result has been shown in Figure 1, where the C&C and all index numbers have been inferred with the demonstrated algorithm.

2.2 Supported attack methods

It's known that Mirai is designed for DDoS attacking purposes. Ten supported attack methods are found in the released Mirai source [8]. Accordingly, ten command codes (e.g. ATK_VEC_xx in Figure 4) are defined in the C&C protocol to deliver attack method types in commands from the controller to bots.

```
BOOL attack_init(void)
{
    int i;

    add_attack(ATK_VEC_UDP, (ATTACK_FUNC)attack_udp_generic);
    add_attack(ATK_VEC_VSE, (ATTACK_FUNC)attack_udp_vse);
    add_attack(ATK_VEC_DNS, (ATTACK_FUNC)attack_udp_dns);
    add_attack(ATK_VEC_UDP_PLAIN, (ATTACK_FUNC)attack_udp_plain);

    add_attack(ATK_VEC_SYN, (ATTACK_FUNC)attack_tcp_syn);
    add_attack(ATK_VEC_ACK, (ATTACK_FUNC)attack_tcp_ack);
    add_attack(ATK_VEC_STOMP, (ATTACK_FUNC)attack_tcp_stomp);

    add_attack(ATK_VEC_GREIP, (ATTACK_FUNC)attack_gre_ip);
    add_attack(ATK_VEC_GREETH, (ATTACK_FUNC)attack_gre_eth);

    //add_attack(ATK_VEC_PROXY, (ATTACK_FUNC)attack_app_proxy);
    add_attack(ATK_VEC_HTTP, (ATTACK_FUNC)attack_app_http);

    return TRUE;
} ? end attack_init ?
```

Figure 4: Ten attack methods found in the released source.

Our analysis shows that the attack methods usually differ in variants in terms of method count, command code numbering, new types of methods, and method implementation. From the point of view of supported attack methods, Mirai variants could be clustered with combinations of 'code-attack_type*n*', where *n* is the version number of a specific attack method. For example, the original version of Mirai could be represented as:

```
{0-atk_udp1, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 6-atk_g
```

Meanwhile, the version of the sample illustrated in Figure 1 (md5=0ae272306d313c6abf1433b85e0a2352) has the following representation:

```
{0-atk_udp1, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 6-atk_g
```

The obvious difference is that command code 10 has different semantics in the two scenarios.

To achieve the combination of ‘code-attack_type*n*’ from a binary sample, we must: (1) extract the attack method codes and their function addresses from the sample; (2) figure out what attack types each function is used for. Accordingly, two analyser programs are designed: one program is to extract {code, attack_function} sequences, while the other is to figure out the extracted attack semantics, as will be introduced in Section 2.2.1.

As shown in Figure 4, Mirai attack methods, together with their command codes, are dynamically installed in a function called ‘attack_init()’. It’s the function named add_attack() that is repeatedly called in attack_init() to save the pairs of {code, attack_function} to a global structure named ‘methods’. Inside the add_attack() function, calloc() is called to allocate memory for the item of {code, attack_function}, and realloc() is called to enlarge the methods by 1 to save the newly allocated item, as shown in Figure 5.

```
static void add_attack(ATTACK_VECTOR vector, ATTACK_FUNC func)
{
    struct attack_method *method = calloc(1, sizeof (struct attack_method));

    method->vector = vector;
    method->func = func;

    methods = realloc(methods, (methods_len + 1) * sizeof (struct attack_method *));
    methods[methods_len++] = method;
}
```

Figure 5: add_attack() function in the released source.

Similar to add_entry() introduced in Section 2.1.1, the add_attack() function may also be inline optimized in some cases, which leads to two versions of binary add_attack(). Fortunately, the two versions share the following characteristics:

1. Composed of a single block.
2. A fixed set of one, or two in the case of inline-optimization, unique functions are called repeatedly.
3. At least one callback function corresponding to the attack method function exists.

The candidate attack_init() functions are found by an IDAPython script according to the above three characteristics. Non-inline and inline versions are considered separately when emulating. The key points of emulating non-inline versions of attack_init() are as follows:

1. NOP’ing all CALL instructions and marking them as breakpoints.
2. Emulating in single-step mode.
3. In the breakpoint handler, saving arguments 1–2 if the second argument points to a valid code area.

When analysing the inline optimization version of code, a trick is used which is inspired by the fact that the newly allocated memory returned from malloc() will be used as the destination operand in the next MEM-WRITE operation to save the code and atk_function. Every time a CALL breakpoint is handled, the return value (e.g. EAX in the case of *Intel* x86 CPU) will be set to a magic value called MAGIC-RET. The key points of emulating an inline optimization version of attack_init() are as follows:

1. NOP’ing all CALL instructions and marking them as breakpoints.
2. Hooking MEM-WRITE instructions.
3. Emulating in single-step mode.
4. In the CALL breakpoint handler, setting the return value as MAGIC-RET.

5. In the MEM-WRITE handler, saving the source operand value as atk_code if the destination operand equals MAGIC_RET.
6. In the MEM-WRITE handler, saving the source operand as atk_function if the destination one equals (MAGIC_RET+4).

An example of the finally found {code, attack_function} pairs is as follows (md5 = 0ae272306d313c6abf1433b85e0a2352):

```
{0_0x8FA0, 1_0xA4C4, 2_0xA988, 8_0x89D8, 3_0xC96C, 4_0xC1F8, 5_0xB998, 10_0xB138, 6_0x9DA8, 7_0x962C, 9_0x8CB
```

The semantics of the found attack functions are inferred using the fingerprint technique introduced in the next section.

2.2.1 Fingerprinting attack functions

Together with the attack method command codes, as many as 25 attack options (e.g. ATK_OPT_PAYLOAD_SIZE, ATK_OPT_PAYLOAD_RAND, ATK_OPT_IP_TOS) are defined in the original Mirai C&C protocol to deliver command details, as shown in Figure 6.

```
#define ATK_OPT_PAYLOAD_SIZE    0   // What should the size of the packet data be?
#define ATK_OPT_PAYLOAD_RAND    1   // Should we randomize the packet data contents?
#define ATK_OPT_IP_TOS          2   // tos field in IP header
#define ATK_OPT_IP_IDENT        3   // ident field in IP header
#define ATK_OPT_IP_TTL          4   // ttl field in IP header
#define ATK_OPT_IP_DF           5   // Dont-Fragment bit set
#define ATK_OPT_SPORT           6   // Should we force a source port? (0 = random)
#define ATK_OPT_DPORT           7   // Should we force a dest port? (0 = random)
#define ATK_OPT_DOMAIN          8   // Domain name for DNS attack
#define ATK_OPT_DNS_HDR_ID      9   // Domain name header ID
//#define ATK_OPT_TCPCC         10  // TCP congestion control
#define ATK_OPT_URG             11  // TCP URG header flag
#define ATK_OPT_ACK             12  // TCP ACK header flag
#define ATK_OPT_PSH             13  // TCP PSH header flag
#define ATK_OPT_RST             14  // TCP RST header flag
#define ATK_OPT_SYN             15  // TCP SYN header flag
#define ATK_OPT_FIN             16  // TCP FIN header flag
#define ATK_OPT_SEQRND          17  // Should we force the sequence number? (TCP only)
#define ATK_OPT_ACKRND          18  // Should we force the ack number? (TCP only)
#define ATK_OPT_GRE_CONSTIP     19  // Should the encapsulated destination address be the same as the target?
#define ATK_OPT_METHOD          20  // Method for HTTP flood
#define ATK_OPT_POST_DATA       21  // Any data to be posted with HTTP flood
#define ATK_OPT_PATH            22  // The path for the HTTP flood
#define ATK_OPT_HTTPS           23  // Is this URL SSL/HTTPS?
#define ATK_OPT_CONNS           24  // Number of sockets to use
#define ATK_OPT_SOURCE          25  // Source IP
```

Figure 6: Attack options defined in Mirai C&C protocol.

Detailed analysis shows it's common for the same functionality to be expressed using different codes across different variants. And some options are usually attack-command-specific, e.g. ATK_OPT_METHOD and ATK_OPT_POST_DATA are only used with ATK_VEC_HTTP (a.k.a. attack_app_http, as shown in Figure 7), while ATK_OPT_GRE_CONSTIP is only used with ATK_VEC_GREIP (a.k.a. attack_gre_ip, as shown in Figure 8) and ATK_VEC_GREETH.

```
void attack_app_http(uint8_t targs_len, struct attack_target *targs, uint8_t opts_len, struct attack_option *opts)
{
    int i, ii, rfd, ret = 0;
    struct attack_http_state *http_table = NULL;
    char *postdata = attack_get_opt_str(opts_len, opts, ATK_OPT_POST_DATA, NULL);
    char *method = attack_get_opt_str(opts_len, opts, ATK_OPT_METHOD, "GET");
    char *domain = attack_get_opt_str(opts_len, opts, ATK_OPT_DOMAIN, NULL);
    char *path = attack_get_opt_str(opts_len, opts, ATK_OPT_PATH, "/");
    int sockets = attack_get_opt_int(opts_len, opts, ATK_OPT_CONNS, 1);
    port_t dport = attack_get_opt_int(opts_len, opts, ATK_OPT_DPORT, 80);
```

Figure 7: Attack options used in attack_app_http().

```
void attack_gre_ip(uint8_t targs_len, struct attack_target *targs, uint8_t opts_len, struct attack_option *opts)
{
    int i, fd;
    char **pkts = calloc(targs_len, sizeof (char *));
    uint8_t ip_tos = attack_get_opt_int(opts_len, opts, ATK_OPT_IP_TOS, 0);
    uint16_t ip_ident = attack_get_opt_int(opts_len, opts, ATK_OPT_IP_IDENT, 0xffff);
    uint8_t ip_ttl = attack_get_opt_int(opts_len, opts, ATK_OPT_IP_TTL, 64);
    BOOL dont_frag = attack_get_opt_int(opts_len, opts, ATK_OPT_IP_DF, TRUE);
    port_t sport = attack_get_opt_int(opts_len, opts, ATK_OPT_SPORT, 0xffff);
    port_t dport = attack_get_opt_int(opts_len, opts, ATK_OPT_DPORT, 0xffff);
    int data_len = attack_get_opt_int(opts_len, opts, ATK_OPT_PAYLOAD_SIZE, 512);
    BOOL data_rand = attack_get_opt_int(opts_len, opts, ATK_OPT_PAYLOAD_RAND, TRUE);
    BOOL gcip = attack_get_opt_int(opts_len, opts, ATK_OPT_GRE_CONSTIP, FALSE);
    uint32_t source_ip = attack_get_opt_int(opts_len, opts, ATK_OPT_SOURCE, LOCAL_ADDR);
```

Figure 8: Attack options used in attack_gre_ip().

Furthermore, different attack methods usually don’t share the same set of attack options, which inspired us to devise a fingerprint technique as follows:

fingerprint(atk_function)={concatenation of used option codes}

For example, the attack_app_http in Figure 6 has a fingerprint of ‘0x15_0x14_0x08_0x16_0x18_0x07’, while attack_gre_ip in Figure 7 has a fingerprint of ‘0x02_0x03_0x04_0x05_0x06_0x07_0x00_0x01_0x13_0x19’.

With the fingerprint technique, not only can we cluster the variants based on their attack method fingerprints, but we are also able to recognize the supported attack methods of new samples with a signature database of {atk_fingerprint, method_name}. The used option codes are all the data that is needed for fingerprinting. They are always referenced in the first, yet large, instruction block of binary attack functions, as shown in Figure 9, by calling the parser functions of attack_get_opt_int()/attack_get_opt_str()/attack_get_opt_ip().

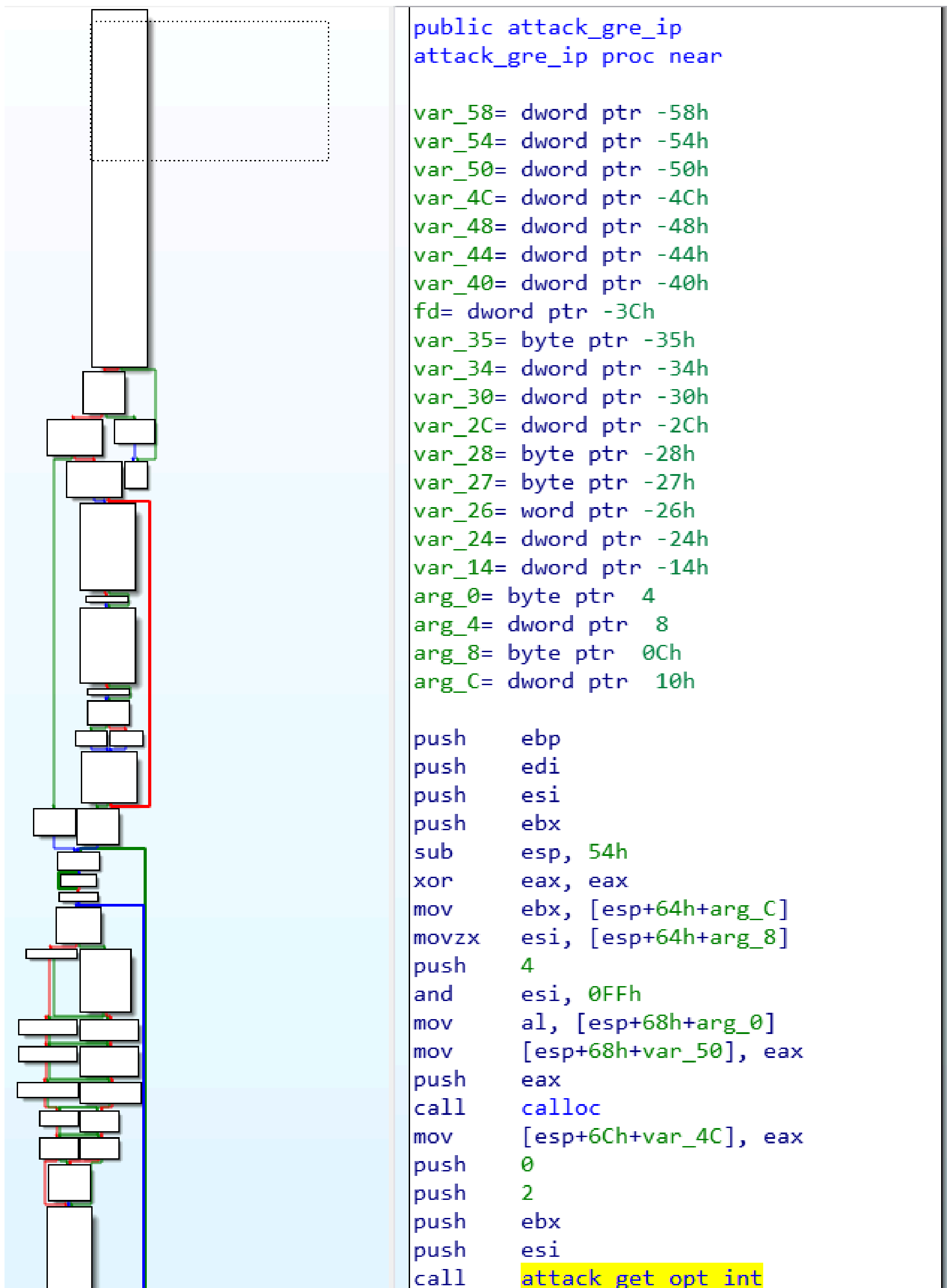

Figure 9: Attack options used in attack_gre_ip().

Detailed analysis shows that all parser functions share the same characteristics: taking four arguments with arguments 1–2 as source data, arg 3 as option code and arg 4 holding a default value. Inspired by the findings, our option code extraction is achieved by emulating the first instruction block of the attack function. The static analysis part is relatively simple: splitting the target attack function into blocks and finding the starting and ending addresses of the first block. A trick to recognize the 'attack_get_opt_' prefixed parser functions in emulation is to set the first and second arguments of the target attack function as two magic values (e.g. MAGIC-VAL1 and MAGIC-VAL2) before emulation starts. The key points of emulation are as follows:

1. Setting arguments 1 and 2 of the target function as two magic values (e.g. MAGIC-VAL1 and MAGIC-VAL2).
2. NOP'ing all the CALL instructions and marking them as breakpoints.
3. Emulating in single-step mode.

4. Saving arguments 3–4 when breakpoints are encountered with arguments 1–2 equal to MAGIC-VAL1–2.

Currently, 43 unique attack method fingerprints have been found from the 32,000+ samples, as illustrated in Appendix B.

2.3 Dictionary of usernames/passwords

It's the scanner module that enables Mirai to infect hundreds of thousands of vulnerable devices in a relatively short period of time. That module can be divided into two parts: (1) the TCP port prober; (2) a telnet brute force attacker with a dictionary of dozens of, default to 62, usernames/passwords. Due to its proven efficiency, the prober is not only retained by most Mirai descendants, but also borrowed by other botnet families [9]. Meanwhile, telnet brute forcing is also kept by most variants, sometimes used together with new exploits [3, 5], which makes it common to find a credential dictionary in Mirai variant samples. Our analysis shows that different variants don't usually share the same set of credentials, as shown in Figure 10. Since new usernames and passwords often indicate new infection vectors, the dictionary changes can be used for variant detection and tracking [10].

```
"9:DEFAULT:b64_TGpXUEJPFg==", "9:DEFAULT:b64_cxJGZ1FuZ1M=", "9:DEFAULT:b64_b1hITFdzZxg=", "10:DEFAULT:DEFAULT", "9:ADMIN:ADMIN", "8:ADMIN:b64_SE8UVUtVFkFU", "8:ADMIN:b64_RFZS
EhUYEBISEg==", "10:ROOT:b64_SVBDQU1/U1QVExUQ", "1:ROOT:VIZXV", "10:b64_V1NUQVJDQU0SEBEV:b64_EhARFRANEBI=", "1:SUPPORT:SUPPORT", "1:USER:USER", "1:GUEST:GUEST", "1:GUEST:b64_E
RITFBU=", "1:ROOT:ADMIN", "1:ADMIN:b64_ERITFA==", "1:ADM:", "1:BIN:", "1:DAEMON:DAEMON", "1:VOLITION:VOLITION", "1:DEFAULT:ANTSLQ", "1:ADMIN:b64_Q2BOVFYSEBAQ"

"4:root:", "8:root:anko", "12:root:hunt5759", "12:telnet:telnet", "7:default:", "12:root:password", "10:admin:admin", "9:root:vizxv", "14:support:support", "11:root:default",
 "17:root:7ujMko0admin", "9:root:12345", "8:user:user", "9:user:12345", "15:default:OxhlwSG8", "15:default:S2fGqNFs"
```

Figure 10: Two examples of extracted usernames and passwords (md5_1: dbba02b2d0ef42d2a1ebbab7f03f37f0; md5_2: 2ff2d4feff4ffcec355f52993ce7b73e).

In Mirai, the scanning task, including the credential dictionary initialization, is done in a function called scanner_init(), which is characterized by having a complex binary control flow graph embedded with a very large instruction block, as illustrated in Figure 11.

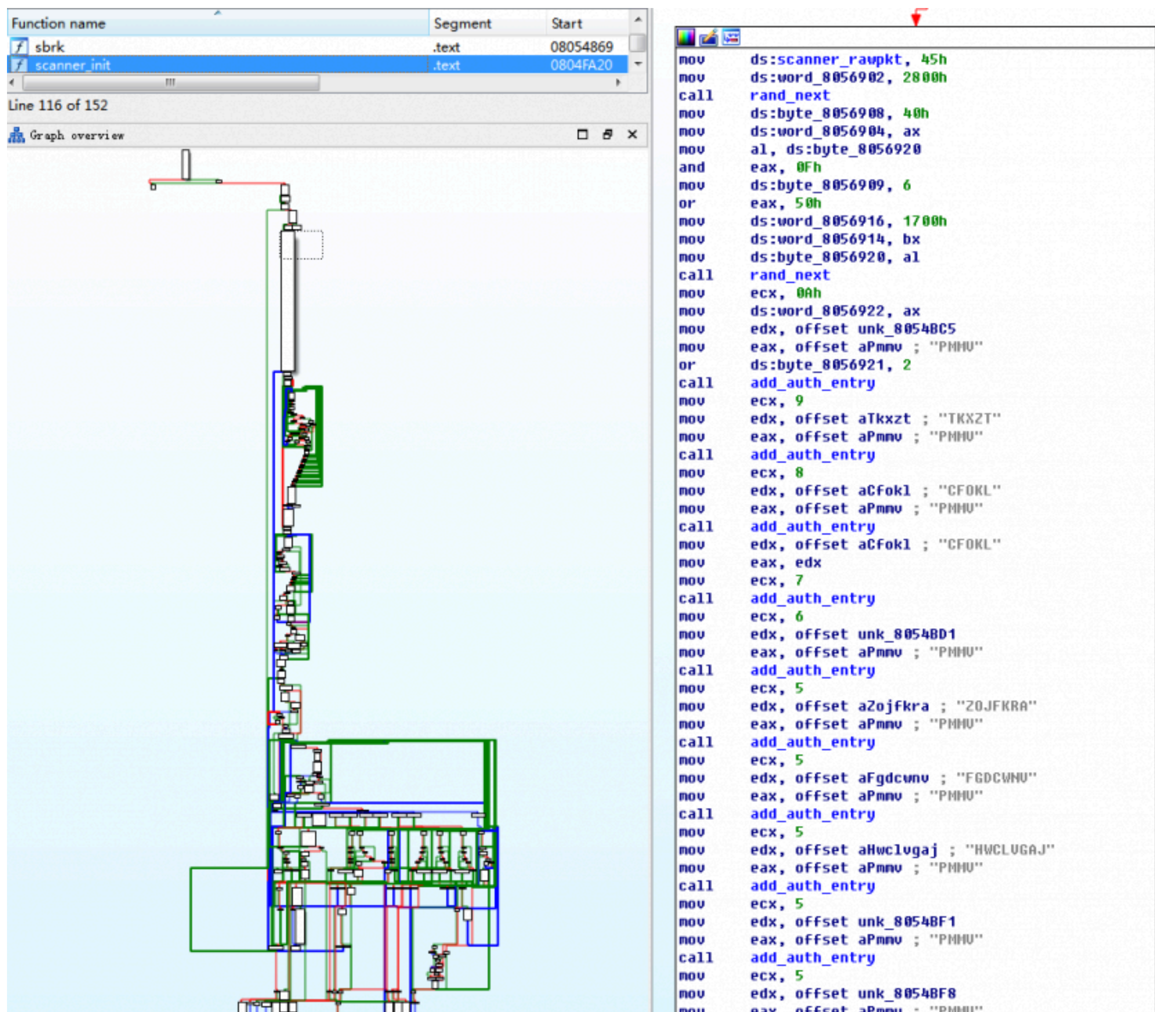

Figure 11: Binary version of scanner_init().

Detailed analysis shows it's the large block that is responsible for installing all the usernames and passwords, which inspires the idea of extracting the credentials by emulating the large block. Candidate blocks are found using an IDAPython script that iterates all binary blocks and picks out those with the following criteria:

1. Having large block size and instruction count.
2. Referencing dozens of global variables which point to ciphertext usernames and passwords.
3. Writing at least five global variables which correspond to IP and TCP fields.
4. Repeatedly calling a unique function (a.k.a. add_auth_entry()).

The found blocks are emulated based on our lightweight emulation framework. The key points of the emulation program are as follows:

1. NOP'ing all the CALL instructions and marking them as breakpoints.
2. Emulating in single-step mode.
3. In the CALL breakpoint handler, saving arguments 1–3 separately as username, password and weight.

The finally extracted usernames and passwords are deciphered with a self-implemented decryption module similar to that introduced in Section 2.1.1. Two examples of the extracted usernames and passwords are illustrated in Figure 10.

3. Variant classification and tracking

While variant classification and tracking is a complex topic, especially in the case of dozens of variants and tens of thousands of samples, in this section we will discuss such schemes with data limited to those introduced in Section 2.

3.1 Configuration-based schemes

Since the configuration database stores most of the running parameters, the variant evolution can be well detected from database changes. For example, the samples illustrated in Figures 12 and 13 share the same C&C and similar configurations, but extra iptables commands are found in the second one, which indicates an exclusive infection and possible infection vectors.

```
CNC=185.246.152.173, port=831, arch=arml, idx_port=1, key=0x37, alg=1
[0x01]: "\x03?", size=2
[0x02]: "H\x8b", size=2
[0x03]: "Oh well...\x00", size=11
[0x04]: "shell\x00", size=6
[0x05]: "enable\x00", size=7
[0x06]: "system\x00", size=7
[0x07]: "SH\x00", size=3
[0x08]: "/bin/busybox WICKED\x00", size=20
[0x09]: "WICKED: applet not found\x00", size=25
[0x0a]: "ncorrect\x00", size=9
[0x0b]: "/bin/busybox ps\x00", size=16
[0x0c]: "/bin/busybox kill -9\x007", size=22
[0x0d]: "ogin\x00", size=5
[0x0e]: "enter\x00", size=6
[0x0f]: "assword\x00", size=8
[0x10]: "/proc/\x00", size=7
[0x11]: "/exe\x00", size=5
[0x12]: "/fd\x00", size=4
[0x13]: "/maps\x00", size=6
[0x14]: "/proc/net/tcp\x00", size=14
[0x15]: "dvrHelper\x00", size=10
[0x16]: "NiGGeR69xd\x00", size=11
[0x17]: "1337SoraLOADER\x00", size=15
[0x18]: "X19I239124UIU\x00", size=14
[0x19]: "14Fa\x00", size=5
[0x1a]: "ccAD\x00", size=5
[0x1b]: "/dev/watchdog\x00", size=14
[0x1c]: "/dev/misc/watchdog\x00", size=19
[0x1d]: "/dev/FTWDT101_watchdog\x00", size=23
[0x1e]: "/dev/FTWDT101\ watchdog\x00", size=24
[0x1f]: "echo '\xc2\xaf\_(\xe3\x83\x84)_/\xc2\xaf Oh hey there... Looks like I might of inected your device.' >> /tmp/wicked.txt\x00", size=99
[0x20]: "echo '\xc2\xaf\_(\xe3\x83\x84)_/\xc2\xaf Oh hey there... Looks like I might of inected your device.' >> /wicked.txt\x00", size=95
[0x21]: "echo '\xc2\xaf\_(\xe3\x83\x84)_/\xc2\xaf Oh hey there... Looks like I might of inected your device.' >> /root/wicked.txt\x00", size=100
[0x22]: "echo '\xc2\xaf\_(\xe3\x83\x84)_/\xc2\xaf Oh hey there... Looks like I might of inected your device.' >> /home/wicked.txt\x00", size=100
[0x23]: "zFFS9ybn0Ccma6nCM92WhLZIwgAmZtH8qWz1vvTGJHJtEU31D87cIVXR6MM9m08P\x00", size=65
[0x24]: "TSource Engine Query\x00", size=21
[0x25]: "/etc/resolv.conf\x00", size=17
[0x26]: "nameserver\x00", size=11
```

Figure 12: Configurations from a sample with md5 = dbba02b2d0ef42d2a1ebbab7f03f37f0.

```
CNC=185.246.152.173, port=38131, c2_in_rsca=true, arch=arml, idx_port=1, key=0x37, alg=1
[0x01]: "\x94\xf3", size=2
[0x02]: "\xc8\x97", size=2
[0x03]: "\xc2\xaf\_(\xe3\x83\x84)_/\xc2\xaf Oh well...\x00", size=25
[0x04]: "shell\x00", size=6
[0x05]: "enable\x00", size=7
[0x06]: "system\x00", size=7
[0x07]: "SH\x00", size=3
[0x08]: "/bin/busybox WICKED\x00", size=20
[0x09]: "WICKED: applet not found\x00", size=25
[0x0a]: "ncorrect\x00", size=9
[0x0b]: "/bin/busybox ps\x00", size=16
[0x0c]: "/bin/busybox kill -9\x007", size=22
[0x0d]: "ogin\x00", size=5
[0x0e]: "enter\x00", size=6
[0x0f]: "assword\x00", size=8
[0x10]: "/proc/\x00", size=7
[0x11]: "/exe\x00", size=5
[0x12]: "/fd\x00", size=4
[0x13]: "/maps\x00", size=6
[0x14]: "/proc/net/tcp\x00", size=14
[0x15]: "dvrHelper\x00", size=10
[0x16]: "NiGGeR69xd\x00", size=11
[0x17]: "1337SoraLOADER\x00", size=15
[0x18]: "X19I239124UIU\x00", size=14
[0x19]: "14Fa\x00", size=5
[0x1a]: "ccAD\x00", size=5
[0x1b]: "/dev/watchdog\x00", size=14
[0x1c]: "/dev/misc/watchdog\x00", size=19
[0x1d]: "/dev/FTWDT101_watchdog\x00", size=23
[0x1e]: "/dev/FTWDT101\ watchdog\x00", size=24
[0x1f]: "iptables -A INPUT -p tcp --destination-port 23 -j DROP\x00", size=55
[0x20]: "iptables -A INPUT -p tcp --destination-port 2323 -j DROP\x00", size=57
[0x21]: "iptables -A INPUT -p tcp --destination-port 22 -j DROP\x00", size=55
[0x22]: "iptables -A INPUT -p tcp --destination-port 8080 -j DROP\x00", size=57
[0x23]: "iptables -A INPUT -p tcp --destination-port 52869 -j DROP\x00", size=58
[0x24]: "iptables -A INPUT -p tcp --destination-port 37215 -j DROP\x00", size=58
[0x25]: "iptables -A INPUT -p tcp --destination-port 53413 -j DROP\x00", size=58
[0x26]: "iptables -A INPUT -p udp --destination-port 53413 -j DROP\x00", size=58
[0x27]: "iptables -A INPUT -p tcp --destination-port 7547 -j DROP\x00", size=57
[0x28]: "killall5 telnet\x00", size=16
[0x29]: "echo '\xc2\xaf\_(\xe3\x83\x84)_/\xc2\xaf Oh hey there... Looks like I might of inected your device.' >> /tmp/wicked.txt\x00", size=99
[0x2a]: "echo '\xc2\xaf\_(\xe3\x83\x84)_/\xc2\xaf Oh hey there... Looks like I might of inected your device.' >> /wicked.txt\x00", size=95
[0x2b]: "echo '\xc2\xaf\_(\xe3\x83\x84)_/\xc2\xaf Oh hey there... Looks like I might of inected your device.' >> /root/wicked.txt\x00", size=100
[0x2c]: "echo '\xc2\xaf\_(\xe3\x83\x84)_/\xc2\xaf Oh hey there... Looks like I might of inected your device.' >> /home/wicked.txt\x00", size=100
[0x2d]: "zFFS9ybn0Ccma6nCM92WhLZIwgAmZtH8qWz1vvTGJHJtEU31D87cIVXR6MM9m08P\x00", size=65
[0x2e]: "TSource Engine Query\x00", size=21
[0x2f]: "/etc/resolv.conf\x00", size=17
[0x30]: "nameserver\x00", size=11
```

Figure 13: Configurations from a sample with md5 = 08abb658c6a293886a8000a31b900e88.

Considering that there are usually dozens of items in a single configuration database, and configurations vary greatly across variants in terms of size and content, it might be not feasible to use all configurations for classification and tracking. Therefore, we devise two schemes which rely only on configuration size, count, and encryption key used.

3.1.1 Clustering based on configuration count and size

This scheme is designed to cluster variants and quickly detect anomalous ones in cases where there are a large number of samples. This is achieved by plotting all the samples according to their configurations' counts and sizes, as shown in Figure 14 where the x-axis represents size while the y-axis shows the count.

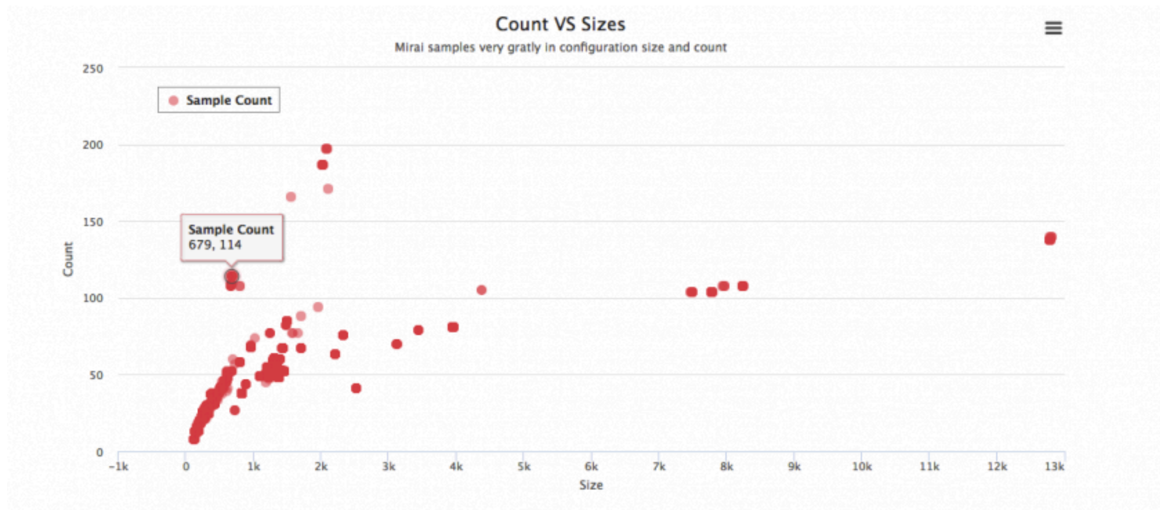

Figure 14: Clustering samples based on configuration count and size.

Two clusters of samples with much larger configuration size can easily be identified in Figure 14. Further analysis shows they can be classified as the same variant, as shown in Table 2. Detailed analysis shows the extraordinarily large configuration size is due to dozens of fake HTTP user-agents added by the author for better HTTP flooding attacks. In fact, the correlated intelligence of branch names, keys and C&Cs strongly suggests that the same underground group is behind the samples.

Criterion	Branch name	Key	C&C	Samples
size > 7400 && count > 100 && count < 120	MIRAI	0x22	cnc.ttoww.com	19
	KYUBI	0x34	cnc.aandy.xyz	4
	MIRAI	0x34	cnc.aandy.xyz	8
	MIRAI	0x34	www.aandy.cf	7
	MIRAI	0x34	www.askjasghasg.ru	16

Table 2: A cluster with very large configuration size.

This clustering scheme can also be used with other schemes. For example, it can be used to cluster samples belonging to the same branch (e.g. MASUTA, OWARI). Clustering results based on size and count can be useful for further analysis.

3.1.2 Clustering based on encryption key

Except for SATORI, which is not discussed in this paper [5], all the Mirai samples we collected share the same encryption algorithm, thus only the encryption key is considered in this paper. Since the key has a space of 2^8, there is a low probability that two variants share the same key coincidentally. Therefore key sharing can be thought of as being caused by code sharing, and can be used to correlate the botnet authors behind the samples.

In total, 31 keys have been detected from our collected samples. Their usage stats show a good long tail distribution of samples, as illustrated in Table 3.

Key	Sample count
0x22	4,755
0x54	3,938
0x3D	553
0x45	542
0x66	204
0x37	163
0x6F	125
0x02	90
0x62	77
0x67	69
0x55	52
0x78	37
0x34	35
0x03	29
0x56	20
0x10	14

0x11	13
0x76	8
0x42	8

Table 3: Top 20 frequently used keys.

According to our analysis, the keys located in the tail area usually indicate a single variant, while frequently used keys indicate a relative in the same family. For example, through the key of 0x54, the branches of Cult, JOSHO and OWARI can be connected, as shown in Table 4.

MD5	Config count	Config size	Branch	C&C
0729b89281c831fc035d56fbf14631da	30	333	Cult	198.134.120.150
23a98fc659982da993e7825eb87bb640	30	340	OWARI	198.134.120.150
2ff2d4feff4ffcec355f52993ce7b73e	30	346	JOSHO	198.134.120.150

Table 4: Connecting Cult, JOSHO and OWARI through the key of 0x54.

The connecting process works as follows:

1. The three branches are connected by the samples sharing the same key.
2. The grouped samples have a similar configuration count and size, which indicates that they probably belong to the same variant.
3. The samples share the same C&C, which strongly suggests that they probably come from the same author(s).

Our analysis shows that such connections can be extended to other branches, with most of them finally verified by the shared C&Cs. Based on that finding, we devise a coarse-grained grouping scheme of 'branch+key' for quickly classifying new samples. In total, 92 groups are produced. With the exception of 'MIRAI_0x22', which stands for the default branches, the other top 10 groups based on their C&Cs are shown in Table 5.

Variant	C&C count
JOSHO+0x54	216
OWARI+0x54	134
Cult+0x54	81
SORA+0x54	69
daddyl33t+0x3D	59
MASUTA+0x45	53
EXTENDO+0x54	21
MIORI+0x54	12
dwickedgod+0x3D	10
Saikin+0x66	9
Katrina+0x67	9

Table 5: Top 10 groups of 'branch+key' based on C&C count.

3.2 Attack-method-based schemes

Since Mirai was designed to launch DDoS attacks, the supported attack methods are a sound basis for variant classification. Based on the attack method fingerprinting technique introduced in Section 2.2, we develop a classification scheme based on the combination of {code, attack-type} pairs extracted from samples, as illustrated in Section 3.2.1.

3.2.1 Combination of supported attack methods

According to our analysis, Mirai variants vary greatly in the supported attack methods in terms of method count, code numbering, and implementation. To quickly distinguish different variants of samples, we devise a coarse-grained classification method based on the combination of supported attack method codes (e.g. 0_1_2_3_4_5_6_7_9_10 for default Mirai samples). Two samples are classified as possibly the same variant for further analysis only when they share the same code combination. The top 10 combinations based on the sample count are shown in Table 6.

Attack method code combination	Count
0_1_2_3_4_5_6_7_9_10	4,488
0_1_2_3_4_5_6_7_8_9_10	3,890
0_1_2_3_4_5_6_7_8	976
0_1_2_3_4_5_6_7_8_9	353
0_1_2_3_6_7_8	138
0_1_2_3_4_5_6_7_9	96
0_1_2_3_4	94
0_1_2_3	75
0_1_2_3_4_5_6_7_9_10_11_12	51
0_1_2	48

Table 6: Top 10 attack method code combinations.

Since it's common for the same code to be assigned to different attack methods in different variants, a more precise classification can be achieved using the combination of code and the corresponding attack method fingerprint. With the help of the fingerprint technique introduced in Section 2.2.1, a total of 126 such unique combinations are found, as shown in Appendix C, where each combination can be thought to represent a variant. It's worth mentioning that the fingerprints have been converted to the method name for better readability with the signature database we built, as shown in Appendix B.

4. Typical variants analysis

To better demonstrate our proposed schemes, in this section we will investigate some popular Mirai variants with the proposed schemes.

4.1 MASUTA

In total, four keys are found to be used in this branch. Their stats are shown Table 7.

Variant	Samples	C&Cs
MASUTA+0x45	351	53
MASUTA+0x02	90	5

MASUTA+0x22	9	1
MASUTA+0x55	8	1

Table 7: Stats on MASUTA samples and C&Cs.

Sample clustering on the largest branch of ‘MASUTA+0x45’ is shown in Figure 15.

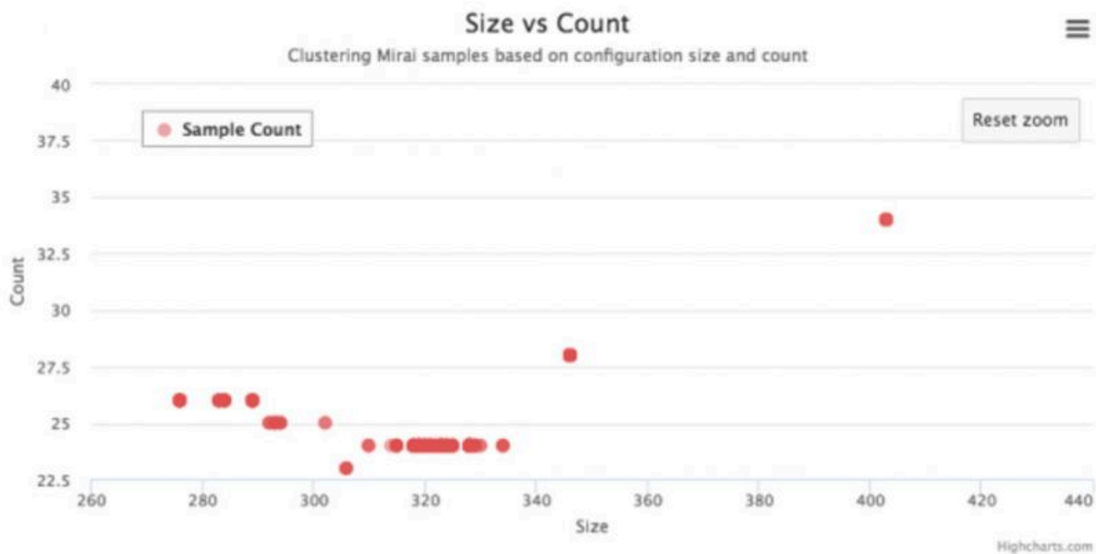

Figure 15: Clustering samples of ‘MASUTA+0x45’ based on configuration.

Both configuration size and count are relatively close. The slight changes in sizes are due to the following reasons:

1. Two different prompt lines are used: ‘gosh that chinese family at the other table sure ate alot’ and ‘The Void’.
2. In some samples the C&Cs are hard coded in resolve_cnc_addr() but old default C&Cs are still kept in configurations.
3. Items of ‘Oaf3’, ‘AbAd’, ‘14Fa’ and ‘WsGA4@F6F’ are added in some samples.

From the point of view of attack methods, as many as 25 combinations of {code, atk_type} are found in all MASUTA samples, while there are eight combinations in the ‘MASUTA+0x45’ branch, as shown in Figure 16.

```
  1  {0-atk_udp3, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn4, 4-atk_tcp_ack7, 5-UNK1, 6-atk_udp_or_gre1, 7-atk_udp_or_gre1, 8-atk_std_or_udp}
  1  {0-atk_udp3, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn4, 4-atk_tcp_ack7, 5-UNK1, 8-atk_std_or_udp}
  3  {0-atk_tcp_syn1, 1-atk_tcp_ack1, 2-atk_tcp_stomp_or_xmas1}
  8  {0-atk_udp1, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 6-atk_gre1, 7-atk_gre1, 9-atk_std_or_udp, 10-atk_http1}
 11  {0-atk_udp_or_gre2, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn5, 4-atk_tcp_ack2, 5-atk_tcp_stomp_or_xmas2, 8-atk_std_or_udp}
 20  {0-atk_udp_or_gre2, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn5, 4-atk_tcp_ack2, 5-UNK1, 6-atk_gre2, 7-atk_gre2, 8-atk_std_or_udp}
 64  {0-atk_udp_or_gre2, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn10, 4-atk_tcp_ack2, 5-atk_tcp_stomp_or_xmas2, 6-atk_gre2, 7-atk_gre2, 8-atk_std_or_udp}
236  {0-atk_udp_or_gre2, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn5, 4-atk_tcp_ack2, 5-atk_tcp_stomp_or_xmas2, 6-atk_gre2, 7-atk_gre2, 8-atk_std_or_udp}
```

Figure 16: Combinations attack methods found 'MASUTA+0x45' samples.

The UNK1 in Figure 16 represents a yet unrecognized fingerprint of ‘0x02_0x03_0x04_0x05_0x07_0x0a_0x0b_0x0c_0x0d_0x0e_0x0f_0x00_0x01_0x06’.

4.2 OWARI

In total, two keys are found to be used in OWARI samples. Their stats are shown in Table 8.

Variant	**Samples**	**C&Cs**
OWARI+0x54	687	146
OWARI+0x66	15	2

Table 8: Stats on OWARI samples and C&Cs.

Sample clustering of OWARI samples is shown in Figure 17.

Figure 17: Clustering OWARI samples based on configuration.

It's interesting that OWARI samples concentrate exactly on four points of (21, 265), (30, 340), (38, 409) and (41, 2524). The slight changes in size are due to the following reasons:

1. Two different prompt lines are used: 'OWARI09123id9i123xd912' and 'Follow twitter.com/1337Wicked'.
2. In the samples with a size of 2524, several automatic download and killer command lines exist, as shown in Figure 18.

Figure 18: Command lines found in OWARI samples with size of 2524.

It's interesting that as many as 12 unique C&Cs are detected from samples with a size of 2424, while they all share the same download server of 185.246.152.173. We think that either they belong to the same group, or the download commands are backdoored by the author (a.k.a. the so-called 'twitter.com/1337Wicked') in his code.

From the point of view of attack methods, only two combinations of {code, atk_type} are found in all OWARI samples:

```
15 {0-atk_udp1, 1-atk_udp_vse1, 2-atk_tcp_syn1, 3-atk_tcp_ack1, 4-atk_gre1, 5-atk_gre1, 6-atk_std_or_udp, 7-a
```

```
642 {0-atk_udp1, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 6-a
```

Detailed analysis shows the first combination corresponds to the branch of 'OWARI+0x66', while the second to 'OWARI+0x54'.

It's worth mentioning that, with the key of 0x66 and the first attack method combination, we successfully connected the 'OWARI+0x66' to another branch named 'kkuuaassaa', where all 44 samples share the same C&C of '46.243.189.109'. We think they are either written by the same author, or there is heavy code sharing.

4.3 WICKED

The WICKED branch became known for including multiple IoT exploits in 2018. In total, 128 samples have been collected (up to May 2018), with only one key of 0x37 found and six C&Cs detected. Sample clustering is shown in Figure 19.

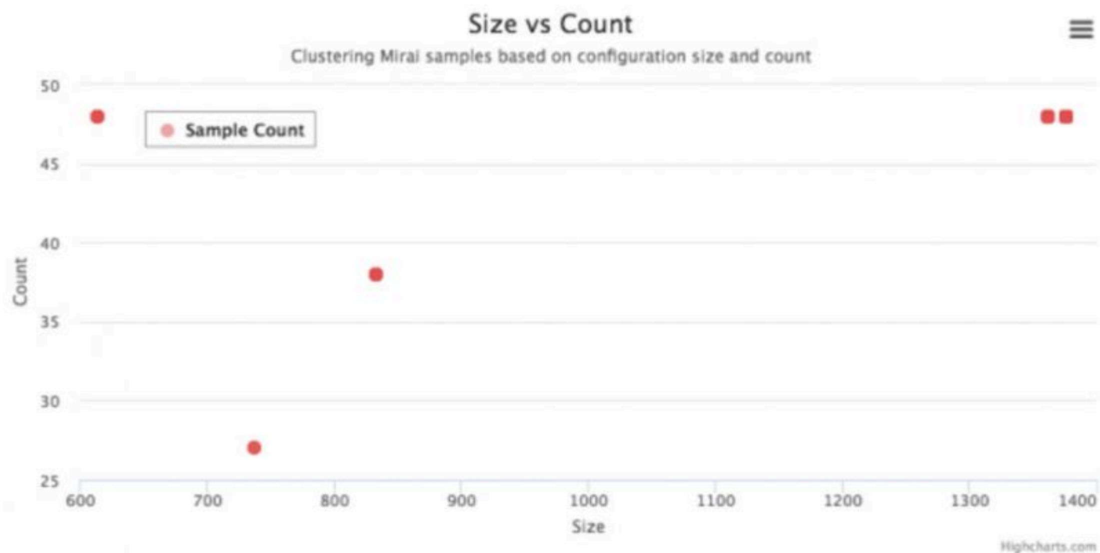

Figure 19: Clustering samples of WICKED based on configuration.

The samples can be clustered into four groups, as shown in Table 9.

(size, count)	**Samples**	**C&C**
(48, 614)	15	104.244.72.82
(27, 737)	4	185.246.152.173
(38, 833)	54	104.236.224.5 104.244.72.82 167.99.220.44 185.189.58.211 185.246.152.173 188.166.63.14
(48, 1362~1376)	55	167.99.220.44 185.246.152.173

Table 9: Stats on WICKED samples.

Given the fact that several C&Cs are shared across clusters, we think that the WICKED samples are probably produced by the same authors. On the other hand, since the IP of 185.246.152.173 is also shared by OWARI samples, the WICKED branch could be connected to that branch.

The size changes in configurations are mainly due to the command lines shown in Figures 20 and 21. Contents shown in Figure 19 indicate an exclusive infection by this branch.

```
iptables -A INPUT -p tcp --destination-port 23 -j DROP\x00
iptables -A INPUT -p tcp --destination-port 2323 -j DROP\x00
iptables -A INPUT -p tcp --destination-port 22 -j DROP\x00
iptables -A INPUT -p tcp --destination-port 8080 -j DROP\x00
iptables -A INPUT -p tcp --destination-port 52869 -j DROP\x00
iptables -A INPUT -p tcp --destination-port 37215 -j DROP\x00
iptables -A INPUT -p tcp --destination-port 53413 -j DROP\x00
iptables -A INPUT -p udp --destination-port 53413 -j DROP\x00
iptables -A INPUT -p tcp --destination-port 7547 -j DROP\x00
```

Figure 20: iptables commands found in WICKED samples.

```
echo '\xc2\xaf\_(\xe3\x83\x84)_/\xc2\xaf Oh hey there... Looks like I might of inected your device.' >> /tmp/wicked.txt\x00
echo '\xc2\xaf\_(\xe3\x83\x84)_/\xc2\xaf Oh hey there... Looks like I might of inected your device.' >> /wicked.txt\x00
echo '\xc2\xaf\_(\xe3\x83\x84)_/\xc2\xaf Oh hey there... Looks like I might of inected your device.' >> /root/wicked.txt\x00
echo '\xc2\xaf\_(\xe3\x83\x84)_/\xc2\xaf Oh hey there... Looks like I might of inected your device.' >> /home/wicked.txt\x00
```

Figure 21: Echo commands found in WICKED samples.

```
 4  {0-atk_udp_or_gre2, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn10, 6-atk_gre2, 7-atk_gre2, 8-atk_std_or_udp, 9-atk_tcp_stomp_or_xmas2}
 7  {0-atk_udp_or_gre2, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn5, 6-atk_gre2, 7-atk_gre2, 8-atk_std_or_udp, 9-atk_tcp_stomp_or_xmas2}
33  {0-atk_udp_or_gre2, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn10, 6-atk_gre2, 7-atk_gre2, 8-atk_std_or_udp}
71  {0-atk_udp_or_gre2, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn5, 6-atk_gre2, 7-atk_gre2, 8-atk_std_or_udp}
```

Figure 22: Combination attack methods found in WICKED samples.

From the point of view of attack methods, combinations of {code, atk_type} are found in WICKED samples, as shown in Figure 22.

5. Conclusion

We have introduced how to extract data including configurations, supported attack methods, and dictionaries of usernames and passwords from Mirai samples, and how to use the extracted data for variant classification and tracking, with different schemes discussed. Some popular Mirai branches have been investigated with our proposed schemes. In the future, we will keep a close eye on the emerging Mirai variants, and will continue researching better classification schemes, e.g. using fuzzy hashing techniques to group samples based on their extracted configurations. We hope our work will help improve the detection and mitigation of Mirai-like threats.

References

[1] Hajime: Analysis of a decentralized internet worm for IoT devices. https://security.rapiditynetworks.com/publications/2016-10-16/hajime.pdf.

[2] IoT_reaper: A Few Updates. http://blog.netlab.360.com/iot_reaper-a-few-updates-en/.

[3] Now Mirai Has DGA Feature Built in. https://blog.netlab.360.com/new-Mirai-variant-with-dga/.

[4] Early Warning: A New Mirai Variant is Spreading Quickly on Port 23 and 2323. https://blog.netlab.360.com/early-warning-a-new-Mirai-variant-is-spreading-quickly-on-port-23-and-2323-en/.

[5] Warning: Satori, a Mirai Branch Is Spreading in Worm Style on Port 37215 and 52869. https://blog.netlab.360.com/warning-satori-a-new-Mirai-variant-is-spreading-in-worm-style-on-port-37215-and-52869-en/.

[6] MMD-0056-2016 – Linux/Mirai, how an old ELF malcode is recycled. http://blog.malwaremustdie.org/2016/08/mmd-0056-2016-linuxMirai-just.html.

[7] Unicorn. https://www.unicorn-engine.org/.

[8] Source Code for IoT Botnet 'Mirai' Released. https://krebsonsecurity.com/2016/10/source-code-for-iot-botnet-Mirai-released/.

[9] ADB.Miner: More Information. https://blog.netlab.360.com/adb-miner-more-information-en/.

[10] Early Warning: A New Mirai Variant is Spreading Quickly on Port 23 and 2323. https://blog.netlab.360.com/early-warning-a-new-mirai-variant-is-spreading-quickly-on-port-23-and-2323-en/.

Appendix A: Sample SHA256 hashes

MD5 hash: 0ae272306d313c6abf1433b85e0a2352

SHA-256 hash: 0c5bc272d13fc05bca15babd83826ca51accf3a8bc0a52f7e0f7f79ea4496280

MD5 hash: dbba02b2d0ef42d2a1ebbab7f03f37f0

SHA-256 hash: 87f49c75ae9aa3138b893a6ff1c316be1c197bed2021ff84bc6a281b25543ee9

MD5 hash: 08abb658c6a293886a8000a31b900e88

SHA-256 hash: dab9ee751d591af93e998b56efa888ef09f50b2e74ab8a6f7b910b00350a866a

MD5 hash: 2db905373ea58920f7dbf9f3e59ba990

SHA-256 hash: 79c3d3b25aba02959ecf734e93b8c162851c11abe81bd7207a16d496ebfa6ab5

MD5 hash: 82358453a5b5be7a54b7013b8f2ec21d

SHA-256 hash: d7ca6f599d37fcfa146b5c044efda4199cb5713fca0984f93301a85b68934c4e

MD5 hash: 0729b89281c831fc035d56fbf14631da

SHA-256 hash: 3af17b130f9b41d5e3645c2622cfe4be5daee0316084cb10c05adf6d60ec1032

MD5 hash: 23a98fc659982da993e7825eb87bb640

SHA-256 hash: f227b9d6f59b27fce5f23551ea15794bd45e26f3eaab44136d6fdf9903992c3b

MD5 hash: 2ff2d4feff4ffcec355f52993ce7b73e

SHA-256 hash: 3cece358fecfc8fbe2e86a1b2c6ae3a0f34d9648cd2306cd734bc717216a728e

MD5 hash: 333d98e27cc885624f073e59fc40dfed

SHA-256 hash: 17a91b2632b625cbd02a009ef64f1faae016de497a7e3b1395e54dc32c8b12d3

Appendix B: Signature of attack methods

Attack method	**Fingerprint**
atk_app_proxy	null
atk_cf	0x08_0x18
atk_gre1	0x02_0x03_0x04_0x05_0x06_0x07_0x00_0x01_0x13_0x19
atk_gre2	0x02_0x03_0x04_0x05_0x06_0x07_0x00_0x01_0x12_0x13
atk_gre3	0x02_0x03_0x04_0x05_0x06_0x07_0x00_0x01_0x12_0x13_0x06
atk_gre4	0x02_0x03_0x04_0x05_0x06_0x07_0x00_0x01_0x12_0x18
atk_gre5	0x02_0x03_0x04_0x05_0x06_0x07_0x00_0x01_0x13_0x06
atk_gre6	0x02_0x03_0x04_0x05_0x06_0x07_0x00_0x01_0x11_0x12
atk_http1	0x15_0x14_0x08_0x16_0x18_0x07
atk_http2	0x14_0x13_0x08_0x15_0x17_0x07

atk_std_or_udp	0x07_0x06_0x00_0x01
atk_tcp_ack1	0x02_0x03_0x04_0x05_0x06_0x07_0x11_0x12_0x0b_0x0c_0x0d_0x0e_0x0f_0x10_0x00_0x01_0x19
atk_tcp_ack2	0x02_0x03_0x04_0x05_0x06_0x07_0x10_0x11_0x0a_0x0b_0x0c_0x0d_0x0e_0x0f_0x00_0x01_0x13
atk_tcp_ack3	0x02_0x03_0x04_0x05_0x06_0x07_0x10_0x11_0x0a_0x0b_0x0c_0x0d_0x0e_0x0f_0x00_0x01_0x13_
atk_tcp_ack4	0x02_0x03_0x04_0x05_0x06_0x07_0x10_0x11_0x0a_0x0b_0x0c_0x0d_0x0e_0x0f_0x00_0x01_0x18
atk_tcp_ack5	0x02_0x03_0x04_0x05_0x06_0x07_0x11_0x12_0x0b_0x0c_0x0d_0x0e_0x0f_0x10_0x00_0x01
atk_tcp_ack6	0x02_0x03_0x04_0x05_0x06_0x07_0x11_0x12_0x0b_0x0c_0x0d_0x0e_0x0f_0x10_0x00_0x01_0x06
atk_tcp_ack7	0x02_0x03_0x04_0x05_0x06_0x07_0x10_0x11_0x0a_0x0b_0x0c_0x0d_0x0e_0x0f_0x00_0x01
atk_tcp_ack_or_synack	0x02_0x03_0x04_0x05_0x06_0x07_0x0f_0x10_0x09_0x0a_0x0b_0x0c_0x0d_0x0e_0x00_0x01_0x12
atk_tcp_stomp_or_xmas1	0x02_0x03_0x04_0x05_0x07_0x0b_0x0c_0x0d_0x0e_0x0f_0x10_0x00_0x01
atk_tcp_stomp_or_xmas2	0x02_0x03_0x04_0x05_0x07_0x0a_0x0b_0x0c_0x0d_0x0e_0x0f_0x00_0x01
atk_tcp_stomp_or_xmas3	0x02_0x03_0x04_0x05_0x07_0x09_0x0a_0x0b_0x0c_0x0d_0x0e_0x00_0x01
atk_tcp_syn1	0x02_0x03_0x04_0x05_0x06_0x07_0x11_0x12_0x0b_0x0c_0x0d_0x0e_0x0f_0x10_0x19
atk_tcp_syn10	0x00_0x02_0x03_0x04_0x05_0x06_0x07_0x10_0x11_0x0a_0x0b_0x0c_0x0d_0x0e_0x0f_0x13
atk_tcp_syn2	0x00_0x02_0x03_0x04_0x05_0x06_0x07_0x10_0x11_0x0a_0x0b_0x0c_0x0d_0x0e_0x0f_0x18
atk_tcp_syn3	0x02_0x03_0x04_0x05_0x06_0x07_0x0f_0x10_0x09_0x0a_0x0b_0x0c_0x0d_0x0e_0x12
atk_tcp_syn4	0x02_0x03_0x04_0x05_0x06_0x07_0x10_0x11_0x0a_0x0b_0x0c_0x0d_0x0e_0x0f
atk_tcp_syn5	0x02_0x03_0x04_0x05_0x06_0x07_0x10_0x11_0x0a_0x0b_0x0c_0x0d_0x0e_0x0f_0x13
atk_tcp_syn6	0x02_0x03_0x04_0x05_0x06_0x07_0x10_0x11_0x0a_0x0b_0x0c_0x0d_0x0e_0x0f_0x13_0x06
atk_tcp_syn7	0x02_0x03_0x04_0x05_0x06_0x07_0x10_0x11_0x0a_0x0b_0x0c_0x0d_0x0e_0x0f_0x18
atk_tcp_syn8	0x02_0x03_0x04_0x05_0x06_0x07_0x11_0x12_0x0b_0x0c_0x0d_0x0e_0x0f_0x10
atk_tcp_syn9	0x02_0x03_0x04_0x05_0x06_0x07_0x11_0x12_0x0b_0x0c_0x0d_0x0e_0x0f_0x10_0x06
atk_udp1	0x02_0x03_0x04_0x05_0x06_0x07_0x00_0x01_0x19
atk_udp2	0x02_0x03_0x04_0x05_0x06_0x07_0x00_0x01_0x11
atk_udp3	0x02_0x03_0x04_0x05_0x06_0x07_0x00_0x01
atk_udp4	0x02_0x03_0x04_0x05_0x06_0x07_0x00_0x01_0x18
atk_udp5	0x02_0x03_0x04_0x05_0x06_0x07_0x00_0x01_0x19_0x11
atk_udp6	0x07_0x06_0x00_0x01_0x10
atk_udp_dns	0x02_0x03_0x04_0x05_0x06_0x07_0x09_0x00_0x08
atk_udp_or_gre1	0x02_0x03_0x04_0x05_0x06_0x07_0x00_0x01_0x12
atk_udp_or_gre2	0x02_0x03_0x04_0x05_0x06_0x07_0x00_0x01_0x13
atk_udp_vse1	0x02_0x03_0x04_0x05_0x06_0x07

atk_udp_vse2	0x02_0x03_0x04_0x05_0x06_0x07_0x11

Appendix C: All recognized combinations of {attack_method_code, attack_type}

```
{0-atk_app_proxy, 1-atk_app_proxy, 2-atk_app_proxy, 3-atk_app_proxy, 4-atk_app_proxy, 5-atk_app_proxy, 6-atk_a
{0-atk_std_or_udp}
{0-atk_tcp_syn1, 1-atk_tcp_ack1, 2-atk_std_or_udp}
{0-atk_tcp_syn1, 1-atk_tcp_ack1, 2-atk_tcp_stomp_or_xmas1}
{0-atk_udp1, 1-atk_std_or_udp, 2-atk_tcp_syn1, 3-atk_tcp_ack1, 4-atk_tcp_stomp_or_xmas1, 5-atk_gre1, 6-atk_udp
{0-atk_udp1, 1-atk_tcp_syn1, 2-atk_tcp_ack1, 3-atk_gre1, 4-atk_gre1, 5-atk_std_or_udp, 6-atk_tcp_stomp_or_xmas
{0-atk_udp1, 1-atk_udp6, 2-atk_tcp_syn1, 3-atk_tcp_ack1, 4-atk_tcp_stomp_or_xmas1, 5-atk_gre1, 6-atk_udp_dns,
{0-atk_udp1, 1-atk_udp_vse1, 2-atk_std_or_udp, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_gre1, 6-atk_gre1, 7-atk_s
{0-atk_udp1, 1-atk_udp_vse1, 2-atk_std_or_udp, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_gre1, 6-atk_gre1, 7-atk_s
{0-atk_udp1, 1-atk_udp_vse1, 2-atk_tcp_syn1, 3-atk_tcp_ack1, 4-atk_gre1, 5-atk_gre1, 6-atk_std_or_udp, 7-atk_s
{0-atk_udp1, 1-atk_udp_vse1, 2-atk_tcp_syn1, 3-atk_tcp_ack1, 4-atk_gre1, 5-atk_gre1, 6-atk_std_or_udp, 7-atk_s
{0-atk_udp1, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 6-atk_gr
{0-atk_udp1, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 6-atk_gr
{0-atk_udp1, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 6-atk_gr
{0-atk_udp1, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 6-atk_gr
{0-atk_udp1, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 6-atk_gr
{0-atk_udp1, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 6-atk_gr
{0-atk_udp1, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 6-atk_gr
{0-atk_udp1, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 6-atk_gr
{0-atk_udp1, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 6-atk_gr
{0-atk_udp1, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 6-atk_gr
{0-atk_udp1, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 6-atk_gr
{0-atk_udp1, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 6-atk_gr
{0-atk_udp1, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 6-atk_gr
{0-atk_udp1, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 6-atk_gr
{0-atk_udp1, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 6-atk_st
{0-atk_udp1, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 9-atk_st
{0-atk_udp1, 1-atk_udp_vse1, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 6-atk_gre1, 7-atk_gre1
{0-atk_udp1, 1-atk_udp_vse2, 2-atk_udp_dns, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 6-atk_gr
{0-atk_udp1, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 6-atk_gre1, 7-atk_gre1, 9-atk_std_or_udp}
{0-atk_udp3, 1-atk_std_or_udp, 2-atk_tcp_syn8, 3-atk_tcp_ack5, 4-atk_tcp_stomp_or_xmas1, 5-atk_udp_or_gre2, 6-
{0-atk_udp3, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn4, 4-atk_tcp_ack7}
{0-atk_udp3, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn8, 4-atk_tcp_ack5, 5-atk_tcp_stomp_or_xmas1, 6-atk_ud
{0-atk_udp3, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn8, 4-atk_tcp_ack5, 5-atk_tcp_stomp_or_xmas1, 6-atk_ud
{0-atk_udp3, 1-atk_udp_vse2, 2-atk_udp_dns, 3-atk_tcp_syn9, 4-atk_tcp_ack6, 5-atk_tcp_stomp_or_xmas1, 6-atk_gr
{0-atk_udp4, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn2, 4-atk_tcp_ack4, 5-atk_tcp_stomp_or_xmas2, 6-atk_gr
{0-atk_udp4, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn2, 4-atk_tcp_ack4, 5-atk_tcp_stomp_or_xmas2, 6-atk_gr
{0-atk_udp4, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn2, 4-atk_tcp_ack4, 5-atk_tcp_stomp_or_xmas2, 6-atk_gr
{0-atk_udp4, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn2, 4-atk_tcp_ack4}
{0-atk_udp4, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn7, 4-atk_tcp_ack4, 5-atk_tcp_stomp_or_xmas2, 6-atk_gr
{0-atk_udp4, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn7, 4-atk_tcp_ack4, 5-atk_tcp_stomp_or_xmas2, 6-atk_gr
{0-atk_udp4, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn7, 4-atk_tcp_ack4, 5-atk_tcp_stomp_or_xmas2, 6-atk_gr
{0-atk_udp4, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn7, 4-atk_tcp_ack4}
{0-atk_udp5, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 6-atk_gr
{0-atk_udp_or_gre1, 1-atk_udp_vse1, 2-atk_tcp_syn3, 3-atk_tcp_ack_or_synack, 4-atk_tcp_ack_or_synack, 5-atk_tc
{0-atk_udp_or_gre2, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn10, 4-atk_tcp_ack2, 5-atk_tcp_stomp_or_xmas2,
{0-atk_udp_or_gre2, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn10, 4-atk_tcp_ack2}
{0-atk_udp_or_gre2, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn10, 6-atk_gre2, 7-atk_gre2, 8-atk_std_or_udp,
{0-atk_udp_or_gre2, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn10, 6-atk_gre2, 7-atk_gre2, 8-atk_std_or_udp}
{0-atk_udp_or_gre2, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn5, 4-atk_tcp_ack2, 5-atk_tcp_stomp_or_xmas2, (
{0-atk_udp_or_gre2, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn5, 4-atk_tcp_ack2, 5-atk_tcp_stomp_or_xmas2, (
{0-atk_udp_or_gre2, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn5, 4-atk_tcp_ack2, 5-atk_tcp_stomp_or_xmas2, (
```

```
{0-atk_udp_or_gre2, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn5, 4-atk_tcp_ack2, 5-atk_tcp_stomp_or_xmas2, 8
{0-atk_udp_or_gre2, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn5, 4-atk_tcp_ack2, 6-atk_gre2, 7-atk_gre2, 8-a
{0-atk_udp_or_gre2, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn5, 4-atk_tcp_ack2, 6-atk_gre2, 7-atk_gre2, 8-a
{0-atk_udp_or_gre2, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn5, 4-atk_tcp_ack2}
{0-atk_udp_or_gre2, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn5, 6-atk_gre2, 7-atk_gre2, 8-atk_std_or_udp, 9
{0-atk_udp_or_gre2, 1-atk_udp_vse1, 2-atk_udp_dns, 3-atk_tcp_syn5, 6-atk_gre2, 7-atk_gre2, 8-atk_std_or_udp}
{1-atk_tcp_syn1, 3-atk_std_or_udp, 4-atk_std_or_udp, 5-atk_tcp_stomp_or_xmas1}
{1-atk_tcp_syn10, 2-atk_tcp_ack2}
{1-atk_tcp_syn5, 2-atk_tcp_ack2}
{1-atk_udp_vse1, 2-atk_tcp_syn3, 3-atk_tcp_ack_or_synack, 4-atk_std_or_udp, 5-atk_std_or_udp, 6-atk_tcp_stomp_
{1-atk_udp_vse1, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 6-atk_gre1, 7-atk_gre1, 9-atk_std_o
{1-atk_udp_vse1, 3-atk_tcp_syn1, 4-atk_tcp_ack1, 6-atk_gre1, 7-atk_gre1, 9-atk_std_or_udp, 10-atk_tcp_stomp_or
{3-atk_tcp_syn1, 4-atk_tcp_ack1, 5-atk_tcp_stomp_or_xmas1, 6-atk_gre1, 7-atk_gre1, 10-atk_http1}
{3-atk_tcp_syn1, 4-atk_tcp_ack1, 6-atk_gre1, 7-atk_gre1, 10-atk_http1}
{3-atk_tcp_syn1, 4-atk_tcp_ack1, 6-atk_gre1, 7-atk_gre1, 9-atk_std_or_udp}
{3-atk_tcp_syn1, 6-atk_gre1, 7-atk_gre1, 10-atk_http1}
{6-atk_gre1, 7-atk_gre1, 10-atk_http1}
{9-atk_std_or_udp}
```

Source: https://www.virusbulletin.com/virusbulletin/2018/12/vb2018-paper-tracking-mirai-variants/#h2-appendix-sample-sha256-hashes