mirai-toushi: Cross-Architecture Mirai Configuration Extractor Utilizing Standalone Ghidra Script

Shun Morishita[1], Satoshi Kobayashi[1], and Eisei Hombu[1]

[1] Internet Initiative Japan Inc.

Abstract

In recent years, IoT malware frequently launches DDoS attacks, causing massive damage to ISPs. Since Mirai and its variants account for the vast majority of IoT malware, security researchers develop configuration extracting tools to understand its characteristics. However, Mirai is built on diverse architectures (e.g., ARM, MIPS, and PowerPC), developing tools is challenging. Indeed, existing tools only support 1 or 2 architectures.

In this study, we utilize Ghidra decompiler and intermediate representation P-Code to reduce architecture-dependent codes, and develop Mirai configuration extractor "mirai-toushi" that supported 8 architectures.

To evaluate mirai-toushi against real-world malware, we applied mirai-toushi to 2,426 samples collected in honeypot/IPS from March 2020 to March 2024. The existing tool extracted 673 tables containing data such as C2 server destinations and DoS parameters, while mirai-toushi extracted 1,743 tables. In addition, mirai-toushi extracted 1,641 password lists. The results show that mirai-toushi can extract Mirai configurations effectively. To be widely used by security researchers, we have made mirai-toushi publicly available on GitHub.

Keywords: Botnet, Malware, IoT, Ghidra, Mirai, mirai-toushi.

1 Introduction

IoT malware had a significant impact in 2016, and even nearly a decade later, in 2025, it still causes severe damage to ISPs and operators [1]. The analysis of malware binaries that infect IoT devices plays a crucial role. For example, it is important for tasks such as identifying attackers' servers, inferring targeted devices, and understanding potential threats.

While IoT malware has been widely studied in the academic field [2], in practical operations, there is a challenge due to the lack of analysis tools compared to Windows malware. The primary reason is that IoT malware infects a wide variety of IoT devices, and its binaries are cross-architecture (e.g., ARM, MIPS, and PowerPC), making tool development more difficult. Additionally, Command & Control (C2) servers in most IoT malware frequently change, and since their addresses are hard-coded within the malware, they cannot be updated. This makes the malware binaries disposable [3], resulting in a large number of samples. How can we efficiently analyze numerous samples across diverse architectures? In practical malware analysis, extracting the malware's "configuration" (config) has been a common approach [4, 5, 6]. Given that in many known malware variants, the code often remains largely unchanged, with the config being the main point of difference. Focusing on the config allows for more efficient analysis by identifying only these differences.

Building on this, we deep dive into specific challenges related to particular malware. By reviewing our honeypot [7], IoTPOT dataset [8, 9], and samples posted on MalwareBazaar [10], we found that the vast majority of the IoT malware samples are Mirai. As analyzing many samples is crucial, we focused on Mirai and its configs. Several tools (extractors) exist for extracting Mirai's configs [11, 12, 13], but they have various limitations (e.g., the number of supported architectures, the types of configs that can be extracted, and the degree of automation).

In order to overcome these challenges, we utilized Ghidra [14] decompiler and intermediate representation P-Code to reduce architecture-dependent codes, and developed Mirai config extractor "mirai-toushi" [15] that supported 8 architectures. For improving the accuracy of the tool, we adopted 2 key methods: (a) First, since decompiled output tends to be inaccurate, we used Ghidra Script's `updateFunction()` to handle the incorrect decompiled output. (b) Second, since malware binaries vary depending on the architecture version and compiler, we tuned the tool using verification malware samples built under various conditions.

To evaluate the effectiveness of the tool with realwolrd malware, we applied mirai-toushi to 2,426 samples collected by honeypots/IPS from May 2020 to May 2024. The existing tool miraicfg [13] extracted 673 tables containing data such as C2 server domain/port, Scan Receiver (SR) domain/port, and DoS parameters, while mirai-toushi extracted 1,743 tables. Additionally, mirai-toushi extracted 1,641 password lists (passlists) used for Telnet scans, confirming the effectiveness of the tool.

To be widely used by security researchers, we first released the version of mirai-toushi used for the experiment on GitHub [15]. Afterward, we made several updates. Specifically, the tool was enhanced to handle specific malware that it previously failed to extract configs in the experiment. In addition, it became capable of extracting new information such as particular C2 servers and DoS attack functions. This is due to the minimal architecture-dependent code in mirai-toushi, which facilitates easier updates and enables future improvements.

Our study contributes both academically and practically, and the key contributions are as follows:

- We developed the Mirai config extractor "mirai-toushi" that supported 8 architectures. It is only target for Mirai, but we believe our development methodology can also be applied to other cross-architecture malware's analysis tools.
- We evaluated mirai-toushi against 2,426 real-world malware samples, and extracted 1,641 passlists and 1,743 tables. It was more effective than the existing tool in terms of the number of configs extracted.
- mirai-toushi is open-sourced [15] and can be utilized for actual operations.

2 Related work

IoT malware has been widely studied from multiple perspectives, including its DDoS attack capabilities, C2 infrastructure, propagation methods, and malware analysis techniques.

The primary goal of IoT malware is to launch DDoS attacks. Therefore, various studies have been conducted on the impact of DDoS attacks and their countermeasures [16, 17]. To gain insight into attacker operations, their C2 infrastructure has been investigated [3, 18, 19, 20]. A large number of infected IoT devices is crucial for carrying out an effective DDoS attack. Consequently, researchers have been monitoring IoT malware's propagation/scanning methods to understand how to infect a wide range of devices [21, 22, 23]. In this field, a decoy system known as a "honeypot" is used to observe scans and collect malware [8, 9, 24, 25]. Once collected, analyzing the malware is essential to understand its behavior, characteristics, and potential threats. Malware analysis is generally divided into 2 techniques: dynamic analysis and static analysis.

Dynamic analysis aims to understand the behavior of malware by executing it and observing its actions. In the case of IoT malware, most studies have been focused on running malware through whole-program emulation [8, 9, 26, 27], using QEMU [28, 29]. Recently, the execution of IoT malware with Qiling [30], an advanced binary emulation framework, has begun to be explored [31, 32].

Static analysis aims to understand the malware's code through reverse engineering, and it may also utilize partial (e.g., code-level and function-level) emulation to analyze its structure and functionality. Since configs often contain the most significant changes in malware, they are a key focus of static analysis, with researchers investigating them to understand malware characteristics [4, 5, 6]. In most malware, these configs are encrypted, making it difficult to examine them from readable strings. The config of our target malware, Mirai, is also encrypted using XOR (We will explain the details in Section 3). Since the XOR key is essentially 1-byte, it is possible to identify the XOR key and decrypt the config by brute-forcing [33, 34, 35]. However, this has the drawback of causing many false positives and being unable to extract information other than strings such as numerical data. Configs can be manually extracted by static analysis, but it is necessary to deal with each malware individually, and the analysis takes a huge amount of time. For this reason, tools for extracting Mirai configs have been developed [11, 12, 13].

decrypting-mirai-configuration-with-radare2 [11] used Radare2 [36] emulation to extract the table of x86 malware. mirai_string_deobfuscation [12] used Binary Ninja [37] intermediate language High Level IL (HLIL) to extract the passlist of ARM malware. These are partially automated, the XOR key and the decrypting function must be identified before running the tool.

miraicfg [13], which was released later, used the disassembly result from Radare2 to extract the table of ARM/x86 malware. This tool does not require manual work to run, achieving full automation.

On the other hand, although the tool has not been made public, existing research [38] conducted a comprehensive investigation of Mirai by analyzing its disassembly result using IDA Pro [39] and performing partial emulation with Unicorn [40]. This research focused on ARM/x86 malware, and investigated not only configs but also DoS attack functions.

3 Mirai config

After the release of Mirai source code [41] in 2016, it has been thoroughly explained [42, 43]. This section reviews the 2 types of encrypted configs in Mirai: passlist and table.

3.1 Passlist

Mirai contains a passlist used for Telnet scans, it is encrypted with an XOR. Passlist is registered by `add_auth_entry()` within `scanner_init()` (Figure 1). `add_auth_entry()` takes 3 arguments: username, password, and weight. Username and password are encrypted with a 4-byte XOR key. The 4-byte XOR key is split into 4 parts (each 1 byte in size) and XORed with corresponding bytes of data. This is equivalent to using a 1-byte XOR key for encryption. For example, if the XOR key is `0xDEADBEEF`, the result will be XORed with `0x22` ($\texttt{byte} \oplus \texttt{0xDE} \oplus \texttt{0xAD} \oplus \texttt{0xBE} \oplus \texttt{0xEF} = \texttt{byte} \oplus \texttt{0x22}$). Weight is used for random selection of the passlist, and it is not encrypted.

```
void scanner_init(void)
{
    add_auth_entry("\x50\x4D\x4D\x56", "\x43\
        x46\x4F\x4B\x4C", 8); // root:admin
    add_auth_entry("\x43\x46\x4F\x4B\x4C", "\
        x43\x46\x4F\x4B\x4C", 7); // admin:
        admin
```

Figure 1: Registration of passlist.

3.2 Table

Mirai contains various configs (table), these values are encrypted with an XOR. The table of original Mirai contains as follows:

- C2 domain/port
- SR domain/port
- String printed to the standard output
- Signature to kill the hostile malware's process
- Command to execute after Telnet login
- Parameter used for DoS attacks

Table is registered as an array by `add_entry()` within `table_init()` (Figure 2). `add_entry()` takes 3 arguments: ID, data, and data length. ID is used as an index in the array. Data is encrypted with a 4-byte XOR key, which is equivalent to being XORed with a 1-byte XOR key, for the same reason as the passlist. Since the XOR keys are defined in different parts of the code, the table XOR key may be different from the passlist XOR key.

```
void table_init(void)
{
    add_entry(TABLE_CNC_DOMAIN, "\x47\x5A\x43\
        x4F\x52\x4E\x47\x0C\x41\x4D\x4F\x22",
        12); // example.com
    add_entry(TABLE_CNC_PORT, "\x22\x35", 2);
        // 23
```

Figure 2: Registration of table.

4 mirai-toushi

4.1 Overview

In this study, we developed a cross-architecture Mirai config extractor "mirai-toushi" [15]. "mirai" is known as future (未来) in Japanese. We appended the Japanese word "toushi" (透視: perspective, 投資: investment) to our tool name.

Malware binaries are built differ not only depending on the architecture type, but also on the architecture version and compiler. To correctly extract configs for a variety of malware binaries, we tuned the tool using verification samples built under various conditions.

An overview of mirai-toushi is shown in Figure 3. We first identify XOR keys used to encrypt passlist/table, and then decrypt configs. It is obvious that passlists are used for login attempts in Telnet scans. On the other hand, tables can be used for various purposes (e.g., C2 domain/port, SR domain/port, and DoS attack parameter). For this reason, we implemented "reference connector" to identify where tables are used.

4.2 Implementation

We implemented a cross-architecture Mirai config extractor utilizing Ghidra [14] decompiler and intermediate representation P-Code. Ghidra is an open source reverse engineering tool released by National Security Agency (NSA), and analysis can be automated using Ghidra Script in Java/Python. Ghidra performs decompilation in a multi-stage process. It first converts binary to assembly, and then converts assembly to P-Code. It decompiles based on P-Code and generates pseudo-code in C. Since P-Code and decompiled C code are not architecture-dependent representations, it allows us to develop the cross-architecture tool.

A comparison of mirai-toushi and existing tools is shown in Table 1. mirai-toushi supports 8 architectures: ARM, MC68000, MIPS, PowerPC, SPARC, SuperH4, x86, and x86_64. In addition, no manual work is required when running it and full automation is achieved.

The implementation of each mirai-toushi feature is described below.

Table 1: Mirai config extractor.

	Reversing Tool	Supported Arch	Passlist	Table	Automation
decrypting-mirai-configuration-with-radare2 [11]	Radare2	x86		✓	
mirai_string_deobfuscation [12]	Binary Ninja	ARM	✓		
miraicfg [13]	Radare2	ARM, x86		✓	✓
mirai-toushi [15]	Ghidra	ARM, MC68000, MIPS, PowerPC, SPARC, SuperH4, x86, x86_64	✓	✓	✓

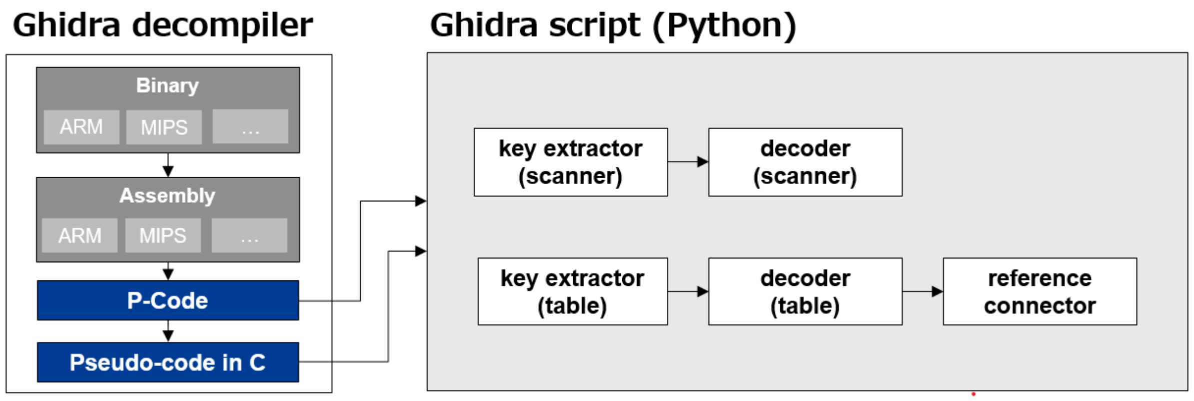

Figure 3: Overview of mirai-toushi.

key extractor (scanner): This feature identifies the XOR key used to encrypt the passlist. In the source code, the 4-byte XOR key is split into 4 parts and XORed 4 times. Due to compiler optimization, these are combined into a single XOR instruction. For this reason, we obtain the decompiled output of each function and identify the instruction that recursively perform a 1-byte XOR on each byte of data. This 1-byte is determined to be the XOR key used to encrypt the passlist.

decoder (scanner): This feature decrypts the passlist using the identified XOR key. Since the passlist is registered by `add_auth_entry()`, we use the decompiled output to identify where `add_auth_entry()` is called. At this time, the number/type of function arguments may not be interpreted correctly, resulting in incorrect decompiled output. To handle this issue, we use Ghidra Script's `updateFunction()` to define the function arguments conrrectly and obtain it. The first argument (username) and the second argument (password) are decrypted using the identified XOR key. The third argument (weight) is not encrypted. Therefore, it is directly converted to a number as plaintext.

key extractor (table): This feature identifies the XOR key used to encrypt the table. It obtains the P-Code of each function and retrieves `INT_XOR` instructions, which are equivalent to XOR. Since XOR is performed 4 times within the target function, it identifies the function that contains this process, and retrieves the 4 bytes that have been XORed. These 4 bytes are determined to be the XOR key used to encrypt the table.

decoder (table): This feature uses the identified XOR key to decrypt the table and calculates the ID to be used in subsequent reference connector. The table is registered by `add_entry()`, which is inlined due to compiler optimization. Therefore, we retrieve the table from `util_memcpy()` called within `add_entry()`. We use Ghidra Script's `updateFunction()` to define the function arguments correctly and obtain the decompiled output. The table data in the second argument is decrypted using the identified XOR key. Since it is most likely the port number of C2/SR, the 2-byte data is decrypted as a numerical value. In addition, we calculate the ID of each data item. Data is stored as an array in the table, and the ID serves as an index. Therefore, the ID can be calculated by subtracting the address of each data item from the base address of the table and dividing it by the data size. The size of the table differed depending on the architecture (Table 2): 6 bytes for MC68000, 8 bytes for other 32-bit architectures, and 16 bytes for x86_64.

$$id = (data\_addr - table\_base\_addr)/data\_size$$

Table 2: Byte size of table memory allocation.

	MC68000	Other 32-bit Arch	x86_64
Data Address	4	4	8
Data Length	2	2	2
Padding	0	2	6
Total	6	8	16

```
table_unlock_val(TABLE_CNC_DOMAIN); //
    Decryption
entries = resolv_lookup(table_retrieve_val(
    TABLE_CNC_DOMAIN, NULL)); // Retrieval
table_lock_val(TABLE_CNC_DOMAIN); // Re-
    Encryption
```

Figure 4: Retrieval of table.

reference connector: This feature identifies the function/address from which data in the table is retrieved. Data is retrieved by `table_retrieve_val()` (Figure 4). Thus, the decompiled output of each function are obtained to identify where `table_retrieve_val()` is called. Since the ID is passed as the first argument, it can be matched with the values calculated by decoder (table) to identify which function/address the data is referenced from.

The analysis results are output in JSON format. Figure 5 shows an example of a passlist output, and Figure 6 shows an example of a table output. Both encrypted configs are decrypted, and the table output includes reference connector result in "refs".

```
{
  "add_auth_entry_func": {
    "name": "add_auth_entry",
    "entrypoint": "0804f8d0",
    "scanner_key": "0x22"
  },
  "scanner_init_func": {
    "name": "scanner_init",
    "entrypoint": "0804fa20",
    "auth_tables_sha256": "0e60e37e94...",
    "auth_tables_count": 23,
    "auth_tables": [
      {
        "user": "root",
        "pass": "admin",
        "weight": 8
      },
      {
        "user": "admin",
        "pass": "admin",
        "weight": 7
      },
```

Figure 5: Example of passlist output.

```
{
  "table_lock_val_func": {
    "name": "table_lock_val",
    "entrypoint": "08050f80",
    "table_key": "0x22",
    "table_original_key": "0xdeadbeef"
  },
  "table_init_func": {
    "name": "table_init",
    "entrypoint": "08051080",
    "tables_sha256": "5c4a784a20...",
    "tables_count": 50,
    "tables_int_count": 2,
    "tables_str_count": 48,
    "tables": [
      {
        "id": 3,
        "type": "str",
        "str_data": "example.com",
        "table_addr": "080565d8",
        "refs": [
          {
            "func": "resolve_cnc_addr",
            "addr": "0804e552"
          }
        ]
      },
      {
        "id": 4,
        "type": "int",
        "int_data": 23,
        "table_addr": "080565e0",
        "refs": [
          {
            "func": "resolve_cnc_addr",
            "addr": "0804e5a9"
          }
        ]
      },
```

Figure 6: Example of table output.

4.3 Tuning

We collected cross-compilers that could be used to build Mirai, and used these to build verification samples from the source code of Mirai and its variants. We then applied our tool to the verification samples and improved the accuracy until it could extract configs correctly.

4.3.1 Collecting cross-compiler

Existing researches [44, 45] have shown that most of cross-compilers in actual use are pre-built toolchains available on the Internet. We investigated the source code of Mirai and its variants published on GitHub and collected cross-compilers that may have actually been used. Out of the 213 source codes, 115 included references to the toolchain, but ultimately, only 4 distinct types of toolchains [46, 47, 48, 49] were identified. From the toolchains, we collected cross-compilers (gcc) for 8 architectures supported by mirai-toushi. At this time, we collected all versions of the compilers when multiple versions of a compiler existed for the same architecture. Finally, 54 types of cross-compilers were collected.

4.3.2 Building verification malware

Using the collected cross-compilers, we built verification samples from the source code of the original Mirai and 3 Mirai variants with different XOR keys:

- `MIRAI` (`0xdeadbeef`/`0x22`)
- `Akiru` (`0xdf7ecadf`/`0xb4`)
- `SORA` (`0xdedefbaf`/`0x54`)
- `WICKED` (`0x1337c0d3`/`0x37`)

We built both samples with and without symbol information. We first built the unstripped samples using `gcc` command of each cross-compiler, and then built the stripped samples using `strip` command. Finally, 370 types of verification samples were output.

4.3.3 Applying to verification malware

We improved the accuracy of our tool using verification samples. The tool was applied to the verification samples, and in cases where configs were not extracted correctly, we modified the tool. By repeating this process, we ultimately confirmed that configs could be correctly extracted from 364 out of the 370 samples, excluding 6 samples that failed to be analyzed by Ghidra.

5 Experiment

To evaluate the effectiveness of the tool against real-world malware, we applied mirai-toushi [15] and miraicfg [13], which can automatically analyze them, to real-world samples collected from honeypots and IPS.

Table 3: Dataset of real-world malware.

Dataset	System	Period	# Malware
IoTPOT Dataset D	Honeypot	05/21/2020–12/31/2022	1,000
IoTPOT Dataset E	Honeypot	01/01/2023–12/31/2023	1,000
IIJ-MALWARE	Honeypot, IPS	03/21/2024–05/31/2024	426

5.1 Dataset

In this experiment, we used real-world samples collected from Yokohama National University's honeypot IoTPOT [8, 9] and "IIJ-MALWARE" collected from IIJ's honeypot [7] and IPS (Table 3). Regarding IoTPOT, we used Dataset D (collected from May 21, 2020 to December 31, 2022) and Dataset E (collected from January 1, 2023 to December 31, 2023). Regarding IIJ-MALWARE, we used samples collected from March 21, 2024 to May 31, 2024.

Since static analysis cannot be performed correctly on packed malware, it must be unpacked before analysis. We identified packers using Detect-It-Easy [50] and confirmed that the majority were UPX [51]. In some malware, `l_info`/`p_info` headers are set to incorrect values, which causes the unpacking process with `upx` command to fail [52]. To handle these malformed headers, we first restored them to the correct headers and then unpacked them using `upx` command.

The collected samples included non-Mirai variants, and even Mirai samples may not contain configs encrypted with a 1-byte XOR. Therefore, we filtered out the samples in advance that may not contain XOR configs. To filter the samples, we used the YARA rule based on the original Mirai source code. Specifically, we filtered the samples to those that contained at least one type of passlist/table signatures encrypted with a 1-byte XOR (`0x01` to `0xFF`). We used 5 passlist signatures (`root`, `admin`, `default`, `user`, and `pass`) and 5 table signatures (`/proc`, `shell`, `enable`, `/bin/busybox`, and `Mozilla`). It is important to note that the signature of either the passlist or table may match that of the other in a malware; therefore, the filtered samples do not necessarily contain both. The YARA rule filtered the samples, leaving 78% of unpacked samples for IoTPOT and 42% for IIJ-MALWARE.

Regarding IoTPOT Dataset D/E, we randomly selected 1,000 samples from the filtered samples. As for IIJ-MALWARE, we selected all 426 filtered samples.

5.2 Applying to real-world malware

The result of applying mirai-toushi/miraicfg to 2,426 samples is shown in Table 4. In the case of mirai-toushi, we also listed the "Passlist ||Table" column, showing the number of samples with at least one of the passlist or table extracted. Since malware may contain only one of them, we included this information.

The result indicates that mirai-toushi extracted a significantly higher number of configs than miraicfg. For mirai-toushi, it extracted 1,641 passlists (68%), and 1,743 tables (72%). On the other hand, miraicfg extracted 673 tables (28%). In addition, when comparing the number of tables extracted from each dataset, mirai-toushi extracted more in all datasets. However, we observed that the number of extractions was low for IIJ-MALWARE. This cause will be discussed in Section 6.1.

The difference in the number of extracted configs between miraicfg and mirai-toushi is due to the difference in the number of supported architectures. miraicfg only supports 2 architectures: ARM (948 samples) and x86 (243 samples). It cannot be run on other architectures (1,240 samples). On the other hand, mirai-toushi supports 8 architectures, and can be run on 2,409 out of 2,426 samples (Table 5). The architectures of the remaining 17 samples were AArch64 (8 samples) and ARC (9 samples). AArch64 is an architecture that we could not be observed from the source code of Mirai variants leaked on GitHub. This rare case reveals that, although limited in number, AArch64 malware exists in the wild. ARC is an architecture that not supported by Ghidra, and mirai-toushi could not support it. However, we confirmed that ARC malware is less common than malware for other architectures.

Table 4: Number of extracted config against real-world malware.

	mirai-toushi [15]			miraicfg [13]
	Passlist	Table	Passlist ||Table	Table
IoTPOT Dataset D (1,000)	862	808	958	339
IoTPOT Dataset E (1,000)	662	726	884	284
IIJ-MALWARE (426)	117	209	247	50
Total (2,426)	1,641	1,743	2,089	673

Table 5: Number of extracted config using mirai-toushi by architecture.

	AArch64 (8)	ARC (9)	ARM (948)	MC68000 (215)	MIPS (476)	PowerPC (251)	SPARC (15)	SuperH4 (220)	x86 (243)	x86_64 (41)
Passlist	0	0	657	161	326	175	3	146	166	7
Table	0	0	609	178	366	198	14	158	210	10
Passlist ||Table	0	0	815	195	409	224	14	196	223	13

5.3 Extracted config

We explain configs extracted by mirai-toushi in this section.

Passlist: 1,641 passlists were extracted in the experiment. We found some differences when compared to the passlists included in the original Mirai. The original contains many passlists targeting IP cameras and DVRs. Meanwhile, real-world malware also contained passlists targeting other types of devices: mobile routers, DSL modems, and ONUs. Moreover, we found notable passlists targeting automatic dog feeders, smart plugs, and smart UPSes.

Botnet name: A botnet name is a string used to identify a botnet. In Mirai, it is determined by the string `MIRAI` in commands like `/bin/busybox MIRAI`, which are executed post-login via a Telnet scan. 1,577 botnet names were identified from 1,743 tables. The most frequently extracted botnet names were `SORA`, `LZRD`, `MIRAI`, `UNSTABLE`, and `DEMONS`.

Port number: 2,355 numerical data were identified from 1,743 tables. In the original Mirai, numerical data is used for the port numbers of C2/SR. Therefore, it is likely that numerical data is also used for these ports in real-world malware. The total number of the ports with values of 1024 or greater was 2,236 (95%). This means that most of the ports were within the ephemeral port range.

In addition, the top 10 most frequently extracted ports totaled 1,267 (54%), with the top 10 accounting for more than half of the total. The top 10 ports are shown in Table 6. We also found that 9 of the top 10 ports matched those used in the source code of Mirai variants leaked on GitHub. (If the port was confirmed in multiple source codes, we listed one of the Mirai variant names in the "Mirai Variant" column.) This means that in real-world malware, the settings in the code are often unchanged, leading to the reuse of ports.

Table 6: Top 10 port number.

#	Port	Usage	Mirai Variant
314	3912	SR	Cosmic-Mirai
312	1312	C2	Cosmic-Mirai
152	9555	SR	Condi-Boatnet
152	3778	C2	Condi-Boatnet
76	17661	?	?
56	1982	SR	Joker-Mirai
53	39284	SR	BeastMode V
52	34712	SR	Amari_Mirai_V2
52	17244	SR	DRACO_1.9_PRIVATE_HYBRID
48	45	C2	Amari_Mirai_V2

Domain/IP address: 621 domains and 136 IP addresses were identified from 1,743 tables. This shows that in real-world malware, domains/IP addresses are often not included in tables, compared to port numbers. In some Mirai variants, we found the C2 domain/IP address is written in the `resolve_cnc_addr()` with plain text, rather than in the table.

Exploit code: The original Mirai only has a Telnet scanner, while recent Mirai variants may include scanners that use exploit code. We confirmed that some variants contain exploit code in the tables. As an example, Mirai variant `DEMONS` contained exploit code targeting a vulnerability in NVMS-9000 DVR.

6 Discussion

6.1 Extraction failure

The result in Section 5.2 confirmed that the number of configs extracted from IIJ-MALWARE was particularly low. The main reason is that IIJ-MALWARE contains many samples built with compiler's optimization level other than `O3` (mainly Mirai variant `MIORI`). In Mirai variant source codes examined in Section 4.3.1, the optimization level was set to `O3` for all except one. Therefore, we built verification samples with `O3` and tuned our tool. As a result, `non-O3` optimization levels were not taken into consideration. We handled this issue after the experiment, and the update will be described in Section 7.2.

In the result of mirai-toushi for each architecture, the lowest extraction rate was x86_64. As for x86_64 malware, 32 out of 41 were IIJ-MALWARE, the extraction rate decreased due to the strong impact of the optimization level, as mentioned above.

6.2 Difference between architectures

Among the samples in the dataset, there were differences in whether they were packed depending on the architecture. Specifically, none of the samples for ARC (9 samples), MC68000 (215 samples), SPARC (15 samples), and SuperH4 (220 samples) were packed. It is considered to be a consequence of UPX, which does not support packing of ELF files for these architectures.

There were also differences in whether malware was stripped. 282 out of 2,426 samples were not stripped, and 245 of these were ARM malware. It is assumed to result from the effects of the ARM cross-compilers. During the build of the verification samples in Section 4.3.2, we observed compile errors occurred when using `strip` command with some ARM cross-compilers. This suggests that the high number of unstripped ARM malware is likely due to the influence of the cross-compilers.

In this way, it became clear that the influence of packers and cross-compilers can cause differences in binaries depending on the architecture. The difficulty of static analysis varies depending on whether packing or stripping. To facilitate the analysis of malware binaries, it is important to develop tools compatible with cross-architectures.

6.3 Limitation

mirai-toushi is not an effective tool against non-Mirai malware. Even for Mirai, it may fail to extract information if the encryption algorithm, compiler, and optimization level are different. In fact, specific Mirai variants, which do not use 1-byte XOR for encryption, are observed nowadays [53, 54].

Furthermore, this tool takes longer to execute than existing tools, which creates limitations when analyzing a large number of malware binaries. Specifically, the execution time of miraicfg per malware is 2-3 seconds, whereas mirai-toushi takes approximately 50 seconds. The reasons it takes time to execute are that the decompilation results are used and reference connector, which is not available in existing tools, has been implemented.

6.4 Non-Mirai malware

As an exception, we confirmed cases where configs could be extracted from non-Mirai malware. In the P2P botnet Mozi collected by IIJ's IPS, mirai-toushi extracted the table with the same content as the original Mirai. This is because Mozi contains part of Mirai source code. In the result of reference connector, DoS attack parameters were retrieved within the malware, but no other items were retrieved. This suggests that C2/SR addresses are not actually used, since the communication protocol is P2P.

7 Release

The version of mirai-toushi [15] used in the experiment was first released on GitHub. We then updated it as described in Section 7.2.

7.1 Usage

mirai-toushi can be run without additional settings or libraries in an environment where Ghidra is installed. It can be run using Jython interpreter executed from Ghidra GUI or Headless analyzer executed from CUI.

Extracted configs can be used for various purposes. For example, it is possible to guess devices targeted by malware from passlists, or to use domains, IP addresses, and port numbers of C2/SR extracted from tables as IoC. In addition, mirai-toushi calculates SHA256 hash values of extracted passlists and tables. This makes it possible to identify whether the malware has the same config from the hash value, even if the malware is from a different architecture. It may also be possible to use unknown hash values to detect new Mirai variants.

7.2 Update

mirai-toushi consists of only 2 Python codes for extracting passlists and tables. Although it supports 8 architectures, it contains little architecture-dependent code. This makes it easier to update the tool compared to existing tools, which require code to be developed for each architecture.

In Section 6.1, we found mirai-toushi does not work well with `non-O3` malware. We updated mirai-toushi so that it can correctly extract configs even from such malware. As we applied the updated version to IIJ-MALWARE (426 samples), the number of extracted passlists increased from 117 to 179, and the number of tables increased from 209 to 293.

In Section 5.3, we found some Mirai variants store the C2 address in `resolve_cnc_addr()`. We developed additional script (`parse_main.py`) so that it can extract this C2 address. Figure 7 shows an example of `parse_main.py` result. Since different malware has different DoS attack functions, we have made it possible to extract information about DoS attack functions using this script.

```
{
  "main_func": {
    "name": "main",
    "entrypoint": "0804df60"
  },
  "resolve_cnc_addr_func": {
    "name": "resolve_cnc_addr",
    "entrypoint": "0804dc40",
    "cnc": "192.0.2.1"
  },
  "attack_init_func": {
    "name": "attack_init",
    "entrypoint": "0804a630",
    "attacks_count": 5,
    "attacks": [
      {
        "vector": 0,
        "name": "attack_tcp_syn",
        "entrypoint": "0804b530"
      },
      {
        "vector": 1,
        "name": "attack_tcp_ack",
        "entrypoint": "0804af90"
      },
```

Figure 7: Example of parse_main.py output.

7.3 Ethical consideration

There is a risk that releasing our paper/tool could allow attackers to take countermeasures. However, Mirai source code itself has already been leaked, and the encryption algorithm is well-known simple XOR. Although there are limitations such as a small number of supported architectures, tools for extracting configs already exist, and we have released a more practical tool. For this reason, we believe that the benefits of widespread use by security researchers outweigh the disadvantage.

8 Conclusion

In this study, we developed a cross-architecture Mirai config extractor called "mirai-toushi" utilizing Ghidra decompiler and intermediate representation P-Code. We applied mirai-toushi to real-world malware, and found that the number of configs extracted was significantly higher than the existing tool, showing the effectiveness of our tool. To be widely used, we have made mirai-toushi publicly available on GitHub.

mirai-toushi is basically an effective tool against Mirai, which uses 1-byte XOR encryption for its configs. Therefore, our future challenge will be to develop a general-purpose IoT malware config extractor that is not dependent on any specific malware/encryption algorithm.

Acknowledgment: We would like to thank the IoT-POT team at Yokohama National University's Yoshioka Laboratory for providing the IoT malware dataset.

Author details

Shun Morishita

Internet Initiative Japan Inc.
Iidabashi Grand Bloom 2-10-2 Fujimi, Chiyoda-ku, Tokyo 102-0071, JAPAN
morishita-sh@iij.ad.jp

Satoshi Kobayashi

Internet Initiative Japan Inc.
Iidabashi Grand Bloom 2-10-2 Fujimi, Chiyoda-ku, Tokyo 102-0071, JAPAN
satoshi-k@iij.ad.jp

Eisei Hombu

Internet Initiative Japan Inc.
Iidabashi Grand Bloom 2-10-2 Fujimi, Chiyoda-ku, Tokyo 102-0071, JAPAN
eisei@iij.ad.jp

References

[1] T. Micro, "Iot botnet linked to large-scale ddos attacks since the end of 2024." `https://www.trendmicro.com/en_us/research/25/a/iot-botnet-linked-to-ddos-attacks.html`. last accessed 2025/04/02.

[2] Q.-D. Ngo, H.-T. Nguyen, V.-H. Le, and D.-H. Nguyen, "A survey of iot malware and detection methods based on static features," ICT express, vol. 6, no. 4, pp. 280–286, 2020.

[3] R. Tanabe, T. Tamai, A. Fujita, R. Isawa, K. Yoshioka, T. Matsumoto, C. Gañán, and M. Van Eeten, "Disposable botnets: examining the anatomy of iot botnet infrastructure," in Proceedings of the 15th International Conference on Availability, Reliability and Security, pp. 1–10, 2020.

[4] JPCERTCC, "Malconfscan." `https://github.com/JPCERTCC/MalConfScan`. last accessed 2025/04/02.

[5] kevoreilly, "Capev2." `https://github.com/kevoreilly/CAPEv2`. last accessed 2025/04/02.

[6] c3rb3ru5d3d53c, "mwcfg." `https://github.com/c3rb3ru5d3d53c/mwcfg`. last accessed 2025/04/02.

[7] M. Saito and T. Kobayashi, "Mitf honeypot support for iot devices," Internet Infrastructure Review (IIR), vol. 36, pp. 10–15, 2017.

[8] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow, "Iotpot: analysing the rise of iot compromises," in 9th USENIX Workshop on Offensive Technologies (WOOT 15), 2015.

[9] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow, "Iotpot: A novel honeypot for revealing current iot threats," Journal of Information Processing, vol. 24, no. 3, pp. 522–533, 2016.

[10] abuse.ch, "Malwarebazaar - user elfdigest." `https://bazaar.abuse.ch/user/5877/`. last accessed 2025/04/02.

[11] 0xd3xt3r, "decrypting-mirai-configuration-with-radare2." `https://github.com/0xd3xt3r/blog-code/blob/master/decrypting-mirai-configuration-with-radare2`. last accessed 2025/04/02.

[12] mrphrazer, "mirai_string_deobfuscation." `https://github.com/mrphrazer/mirai_string_deobfuscation`. last accessed 2025/04/02.

[13] FernandoDoming, "miraicfg." `https://github.com/FernandoDoming/miraicfg`. last accessed 2025/04/02.

[14] N. S. Agency, "Ghidra." `https://ghidra-sre.org/`. last accessed 2025/04/02.

[15] IIJ, "mirai-toushi." `https://github.com/iij/mirai-toushi`. last accessed 2025/04/02.

[16] M. De Donno, N. Dragoni, A. Giaretta, and A. Spognardi, "Ddos-capable iot malwares: comparative analysis and mirai investigation," Security and Communication Networks, vol. 2018, no. 1, p. 7178164, 2018.

[17] R. Vishwakarma and A. K. Jain, "A survey of ddos attacking techniques and defence mechanisms in the iot network," Telecommunication systems, vol. 73, no. 1, pp. 3–25, 2020.

[18] A. Davanian, A. Darki, and M. Faloutsos, "Cnchunter: An mitm-approach to identify live cnc servers," Black Hat USA, 2021.

[19] A. Davanian and M. Faloutsos, "Malnet: A binary-centric network-level profiling of iot malware," in Proceedings of the 22nd ACM Internet Measurement Conference, pp. 472–487, 2022.

[20] A. Davanian, M. Faloutsos, and M. Lindorfer, "C2miner: Tricking iot malware into revealing live command & control servers," in Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, pp. 112–127, 2024.

[21] S. Torabi, E. Bou-Harb, C. Assi, E. B. Karbab, A. Boukhtouta, and M. Debbabi, "Inferring and investigating iot-generated scanning campaigns targeting a large network telescope," IEEE Transactions on Dependable and Secure Computing, vol. 19, no. 1, pp. 402–418, 2020.

[22] A. A. Al Alsadi, K. Sameshima, J. Bleier, K. Yoshioka, M. Lindorfer, M. Van Eeten, and C. H. Gañán, "No spring chicken: quantifying the lifespan of exploits in iot malware using static and dynamic analysis," in Proceedings of the 2022 ACM on Asia conference on computer and communications security, pp. 309–321, 2022.

[23] A. A. Al Alsadi, K. Sameshima, K. Yoshioka, M. Van Eeten, and C. H. Gañán, "Bin there, target that: Analyzing the target selection of iot vulnerabilities in malware binaries," in Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses, pp. 513–526, 2023.

[24] M. Wang, J. Santillan, and F. Kuipers, "Thingpot: an interactive internet-of-things honeypot," arXiv preprint arXiv:1807.04114, 2018.

[25] S. Kato, R. Tanabe, K. Yoshioka, and T. Matsumoto, "Adaptive observation of emerging cyber attacks targeting various iot devices," in 2021 IFIP/IEEE International Symposium on Integrated Network Management (IM), pp. 143–151, IEEE, 2021.

[26] D. Uhrıcek, "Lisa–multiplatform linux sandbox for analyzing iot malware," 2020.

[27] A. Darki and M. Faloutsos, "Riotman: a systematic analysis of iot malware behavior," in Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies, pp. 169–182, 2020.

[28] F. Bellard, "Qemu, a fast and portable dynamic translator.," in USENIX annual technical conference, FREENIX Track, pp. 41–46, California, USA, 2005.

[29] T. Q. P. Developers, "Qemu." `https://www.qemu.org/`. last accessed 2025/04/02.

[30] Q. F. project, "Qiling framework." `https://qiling.io/`. last accessed 2025/04/02.

[31] GDATAAdvancedAnalytics, "Qiliot." `https://github.com/GDATAAdvancedAnalytics/Qiliot`. last accessed 2025/04/02.

[32] T. Ljucovic, "Destructive iot malware emulation ‐part 1 of 3 ‐environment setup." https://cyber.wtf/2024/03/28/destructive-iot-malware-emulation-part-1-of-3-environment-setup/. last accessed 2025/04/02.

[33] D. Stevens, "Xorsearch & xorstrings." `https://blog.didierstevens.com/programs/xorsearch/`. last accessed 2025/04/02.

[34] decalage2, "Balbuzard." `https://github.com/decalage2/balbuzard`. last accessed 2025/04/02.

[35] srozb, "mirai-utils." `https://github.com/srozb/mirai-utils`. last accessed 2025/04/02.

[36] radare org, "Radare2." `https://rada.re/`. last accessed 2025/04/02.

[37] V. 35, "Binary ninja." `https://binary.ninja/`. last accessed 2025/04/02.

[38] Y. Liu and H. Wang, "Tracking mirai variants," Virus Bulletin, pp. 1–18, 2018.

[39] Hex-Rays, "Ida pro." `https://hex-rays.com/ida-pro`. last accessed 2025/04/02.

[40] U. engine project, "Unicorn." `https://www.unicorn-engine.org/`. last accessed 2025/04/02.

[41] jgamblin, "Mirai-source-code." `https://github.com/jgamblin/Mirai-Source-Code`. last accessed 2025/04/02.

[42] M. Saito, M. Negishi, T. Kobayashi, T. Nagao, H. Suzuki, M. Kobayashi, H. Nashiwa, M. Kobayashi, and Y. Suga, "Mirai botnet detection and countermeasures," Internet Infrastructure Review (IIR), vol. 33, pp. 4–29, 2016.

[43] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, et al., "Understanding the mirai botnet," in 26th USENIX security symposium (USENIX Security 17), pp. 1093–1110, 2017.

[44] S. Akabane and T. Okamoto, "Identification of library functions statically linked to linux malware without symbols," Procedia Computer Science, vol. 176, pp. 3436–3445, 2020.

[45] S. Akabane and T. Okamoto, "Identification of toolchains used to build iot malware with statically linked libraries," Procedia Computer Science, vol. 192, pp. 5130–5138, 2021.

[46] E. Andersen, "uclibc toolchain 0.9.30.1." `https://www.uclibc.org/downloads/binaries/0.9.30.1/`. last accessed 2025/04/02.

[47] ibiblio, "Slitaz." `http://distro.ibiblio.org/slitaz/sources/packages/c/`. last accessed 2025/04/02.

[48] R. Landley, "Aboriginal linux 1.2.6." `https://landley.net/aboriginal/downloads/old/binaries/1.2.6/`. last accessed 2025/04/02.

[49] R. Landley, "Aboriginal linux 1.4.5." `https://landley.net/aboriginal/downloads/old/binaries/1.4.5/`. last accessed 2025/04/02.

[50] horsicq, “Detect-it-easy.” `https://github.com/horsicq/Detect-It-Easy`. last accessed 2025/04/02.

[51] T. U. Team, “Upx: the ultimate packer for executables.” `https://upx.github.io/`. last accessed 2025/04/02.

[52] JPCERTCC, “Anti-upx unpacking technique.” `https://blogs.jpcert.or.jp/en/2022/03/anti_upx_unpack.html`. last accessed 2025/04/02.

[53] H. Wang, Acey9, and Alex.Turing, “Mirai.tbot uncovered: Over 100 groups and 30,000+ infected hosts in a big iot botnet.” `https://blog.xlab.qianxin.com/mirai-tbot-en/`. last accessed 2025/04/02.

[54] H. Wang, daji, Alex.Turing, and Acey9, “Botnets never die: An analysis of the large scale botnet airashi.” `https://blog.xlab.qianxin.com/large-scale-botnet-airashi-en/`. last accessed 2025/04/02.