## 1 Supplementary details of the empirical study

This section provides some additional supplementary about the empirical research process.

### 1.1 Analysis framework

As mentioned in § 2.2, Figure 1 displays the multi-level analysis framework we adopted in our empirical study of IoT malware. This framework encompasses a detailed analysis process and the tools utilized, divided into three levels. Initially, by uploading samples to VirusTotal, we obtained a generated preliminary analysis report. Subsequently, using this report as a reference, we conducted thorough manual analysis on the corresponding IoT malware samples. At this stage, we first employed the LISA sandbox for dynamic analysis to capture the anomalous behavior of the IoT malware. Based on the results from the LISA sandbox, we further engaged in reverse engineering using Ghidra, thus conducting an in-depth manual analysis for IoT malware at the code level. During the manual analysis stage, our focus was on verifying the accuracy of the VirusTotal report and identifying the fine-grained malicious behaviors that were not reported. Following the new findings from the manual analysis, we refined and supplemented the initial report. It is important to note that, to ensure the accuracy and reliability of our analysis, we referred to the ATT&CK framework developed by MITRE. The ATT&CK framework meticulously defines the tactics, techniques, and procedures (TTPs) attackers may employ, serving as a knowledge base designed to aid security teams in understanding the behavioral patterns of threat actors, thereby enhancing the identification, detection, and defense against attacks.

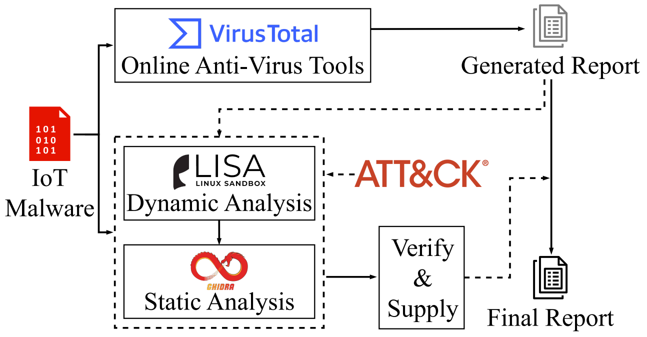

**Figure 1.** IoT malware analysis framework

### 1.2 Distribution of IoT malware by family

As mentioned in § 2.1, Table 1 shows the distribution of malicious families to which the 5K samples belongs. From the table, it can be seen that in the field of IoT malware, Gafgyt and Mirai occupy an absolute dominant position.

| Rank | Label(AVClass) | Count(%) |
|---|---|---|
| 1 | Gafgyt | 3499(69.6) |
| 2 | Mirai | 1168(22.9) |
| 3 | Tsunami | 87(1.70) |
| 4 | Hajime | 9(<1) |
| 5 | Xorddos | 4(<1) |
| 6 | Lightaidra | 3(<1) |
| 7 | Chinaz | 2(<1) |
| 8 | Rudedevil | 2(<1) |
| 9 | Ditertag | 2(<1) |
| 10 | Setag | 2(<1) |
| 11 | Multiverze | 2(<1) |
| 12 | Ddostf | 2(<1) |
| 13 | Dofloo | 2(<1) |
| 14 | Chacha | 1(<1) |
| 15 | Ares | 1(<1) |
| 16 | Bluteal | 1(<1) |
| 17 | Presenoker | 1(<1) |
| 18 | Zpevdo | 1(<1) |
| | Unknown | 312(6.1) |
| | Total | 5101(100) |

**Table 1.** Distribution of IoT malware by family

### 1.3 Distribution of sample scale of the 5K samples

As mentioned in § 2.4, Figure 2 shows the distribution of 5K samples scale. From the figure, it can be observed that the scale distribution of existing IoT malware is relatively concentrated, and samples with extreme sizes are relatively rare. Therefore, when facing adversarial attacks, especially methods that involve adding redundant nodes on a large scale to construct extreme samples far from the norm, existing masked graph approaches may struggle to adapt, leading to performance degradation issues.

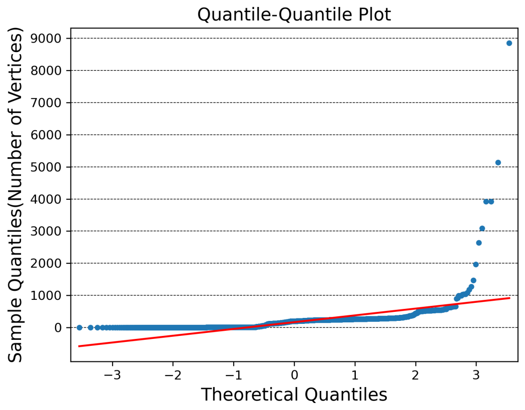

**Figure 2.** Distribution of sample scale of the 5K samples

### 1.4 Edge node example

As illustrated in the figure3, during our analysis, we conduct an in-depth analysis of IoT malware by tracing the function call chains within the function call graph. Upon reaching the *gethostbyname* function node within our analytical process, we are able to delineate the malicious behavior of the malware. In this context, the internal implementation details of the function are not pivotal for understanding the logic behind the malicious behavior, as our primary focus lies on its core functionality, namely, the domain name resolution capabilities it provides for the malware. Therefore, we are not concerned with the specific implementation details of how such functions perform their standard operations; our focus is solely in the role they play and the functionalities they provide within the context of malicious software behavior.

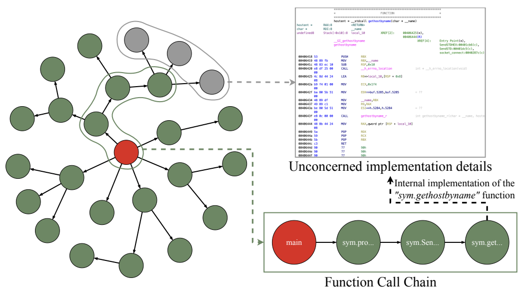
Unconcerned implementation details
Internal implementation of the "sym.gethostbyname" function
main
sym.pro...
sym.Sen...
sym.get...
Function Call Chain

**Figure 3.** Careless implementation details

## 2 Summary of empirical study

In this section, we provide some summarized findings from empirical research. including: (1) the summary and classification of fine-grained malicious behaviors in IoT malware, as shown in table 4; and (2) a summary of the overview and implementation mechanisms of four fine-grained malicious behaviors during the lurking stage of IoT malware, details as follows:

- **Persistence** is the behavior adopted by malware to ensure its long-term existence and regular operation on the target system device, thereby maintaining prolonged control over the victim's system.
  **Implementation -** The first implementation method is auto-start through boot or login initialization scripts, such as adding or modifying scripts in the "*/etc/init.d*". These scripts are executed automatically at system startup or user login or by modifying the rc auto-start scripts in the system to implement malware auto-start. The second is creating or modifying Launch Daemons in the Linux system, making the malware run as a background service, thus achieving persistence. The third is establishing persistence by setting up scheduled tasks, such as modifying the cron task scripts, to execute malicious activities and ensure their continuous operation regularly.
- **Privilege Escalation** is the behavior of obtaining higher-level system permissions than initially granted in order to execute more destructive or theft activities.
  **Implementation -** The first implementation method is the abuse elevation control mechanism, where malware executes operations of users or groups with higher privileges by *setuid* and *setgid*. The second method is exploiting *sudo* and *sudo caching*, such as by modifying the sudoers file, which allows the malware to grant itself higher permissions to perform actions that are normally not executable. The third method uses file permission management commands (such as *chown* or *chmod*) to modify the permissions of files or directories. This enables attackers to read, modify, or execute originally restricted files. The fourth method is leveraging *ptrace* system calls, where malware injects malicious code into a known process with higher privileges and executes code within that process context, thereby indirectly elevating its permissions. Additionally, this method can also help malware evade process-based monitoring.
- **Deception** refers to malware hiding its true intentions or identity by disguising or imitating benign software to deceive users or defense systems.
  **Implementation -** The first implementation method is disguising legitimate software or system files, thus deceiving users and systems. The second method is by changing file extensions or hiding the real extensions and modifying file properties to disguise them as normal files, achieving the purpose of deception. The third method is modifying process parameters to disguise its real purpose, such as changing command-line parameters to deceive and confuse users and systems.
- **Defense Evasion** encompasses various strategies and methods malware uses to avoid being detected by security software and to remain undetected and unblocked in the victim's system.
  **Implementation -** The first implementation method is to automatically hide or modify its core malicious behavior when a debugger is detected to prevent analysis and detection. The second method involves erasing attack traces, such as using commands like "history -c" or "rm /.bash_history" to clear command history or deleting system log files in the "/var/log/" directory. The third method embeds the payload in other files through encryption algorithms, making detecting and analyzing the malware more challenging. The fourth method uses the *sleep* command or system scheduling functions to delay or hide the execution of core malicious functionalities, thereby evading sandbox environments with runtime limitations. Additionally, against advanced sandboxes with time acceleration features, malware often evades detection by assessing the expected difference in environment timestamps before and after executing the sleep function.

## 3 Overview of using algorithms

In this section, detailed descriptions of the 14 basic classification algorithms 2 and 16 multi-label classification algorithms 3 used in the paper are provided, details as follows:

| Algorithm | Acronym | Type |
|---|---|---|
| C4.5 Decision Tree | J48 | trees |
| Reduced Error Pruning Tree | REPTree | trees |
| Random Forest | RF | trees |
| Logistic Model Trees | LMT | trees |
| Logistic Regression | Logistic | functions |
| Sequential Minimal Optimization | SMO | functions |
| K-Nearest Neighbor (KNN) | IBk | lazy |
| Naive Bayes | NaiveBayes | bayes |
| JRip Rule Learning | JRip | rules |
| One Rule | OneR | rules |
| Partial Decision Trees Rule Learning | PART | rules |
| Adaptive Boosting | AdaBoost | meta |
| Bootstrap Aggregating | Bagging | meta |
| Stacked Generalization | Stacking | meta |

**Table 2.** Basic Classification Algorithm

| Algorithm | Acronym | Type |
|---|---|---|
| Bayesian Classifier Chains | BCC | PT |
| Back Propagation Neural Network | BPNN | AA |
| Binary Relevance | BR | PT |
| Classifier Chains | CC | PT |
| Conditional Dependency Networks | CDN | PT |
| Conditional Dependency Trellis | CDT | PT |
| Classifier Trellis | CT | PT |
| Fourclass Pairwise | FW | PT |
| HASEL | HASEL | PT |
| Laber Powerset | LC | PT |
| Classifier Chains with Monte Carlo optimization | MCC | AA |
| Pruned Sets with Threshold | PSt | PT |
| Pruned Sets | PS | PT |
| Random k-Labelsets | RAkEL | PT |
| Random k-Label Disjoint Pruned Sets | RAkELd | PT |
| Ranking and Threshold | RT | PT |

**Table 3.** Multi-Label Classification Algorithm

Received 16 February 2024; revised 23 February 2024; accepted 1 April 2024

| **Stage** | **Behaviors** | **Method** | **Attack Chain Extraction** |
|---|---|---|---|
| Preparation | Scanning | Scan port | Port scanning ⟶ Port status monitoring ⟶ Vulnerability detection ⟶ Service identification ⟶ Record scan results |
| | | Use vocabulary scanning | Build vocabulary ⟶ Use vocabulary for brute force attempts |
| | Automated Collection | Read host information | Read host hardware and software information—send to server |
| | Initial Access | Default Credentials | Get default credentials ⟶ try to log in |
| | Establish communication channels | C&C | Set the C&C server address⟶Set the C&C server port⟶Try to connect to the C&C server |
| Lurking | Persistency | With the help of initialization script file | Read the boot or login initialization script file ⟶ modify the script file ⟶ automatically execute the initialization script file |
| | | Set up daemon | Create or modify property list file ⟶ Reload Systemd Manager ⟶ Enable and start the service ⟶ Register the malware as a background service |
| | | Modify the scheduled task script file | Read the system scheduled task script file ⟶ modify the system scheduled task script file ⟶ add the regularly executed task of the malware ⟶ restart the scheduled task service |
| | Privilege escalation | With system call | Execute relevant system call functions ⟶ change the effective user ID and effective group ID ⟶ perform operations for high-privileged users or groups |
| | | Modify configuration file | Read the rights management configuration file ⟶ modify the rights management configuration file ⟶ perform normally restricted operations |
| | | With the help of related commands | Execute file permission management commands ⟶ change the permissions of files or directories ⟶ read, modify or execute originally restricted files |
| | | process injection | Obtain a higher-privileged process ⟶ inject malicious code into that process ⟶ execute the malicious code in the context of that process |
| | Deception | Modify file name | Get the file name and path ⟶ modify the file name and path |
| | | Modified file attributes | Get file extension ⟶ modify or hide file extension |
| | | Modify process parameters | Get process parameters ⟶ modify process parameters |
| | Defense Evasion | Hide malicious behavior | Monitor the process list or system log ⟶ Detect whether there is a debugger in the system ⟶ Use judgment methods to hide or change core malicious behaviors in advance |
| | | Clear malicious history | Execute the command to clear the command history ⟶ Get the system log file path ⟶ Clear the system log file |
| | | Encrypted payload | Encrypt payload using algorithm ⟶ embed other legitimate files |
| | | Delay execution of malicious actions | Execute the sleep command to delay the execution of the core malicious behavior ⟶ Calculate the environment timestamps before and after command execution ⟶ Determine whether you are in a sandbox ⟶ Hide or change the core malicious behavior |
| Attack | DDoS | capacity exhaustion attack | Generate a large number of data packets (random or forged data) ⟶ send data packets to the target service in a loop ⟶ consume the target network bandwidth/resources |
| | | protocol attack | Construct a malicious request (forge the source address to be the target address) ⟶ Send requests to the target service in a loop at high frequency ⟶ Service response floods the target |
| | | Application layer attack | Construct legitimate application layer requests ⟶ send a large number of requests to the target service in a loop ⟶ consume application layer resources |
| | Data Theft | Data theft | Read local file ⟶ Get sensitive information ⟶ Post to The remote server |
| | Destruction | Data destruction | Read local file ⟶ Delete |
| | | Data tampering | Read local file ⟶ Modified |

**Table 4.** A detailed summary table of malicious behaviors