Information Engineering Express
International Institute of Applied Informatics
2025, Vol. 11, No. 1, IEE885

YODA: Unified Middleware for IoT Firmware Static Analysis Tools

Minami Yoda [*], Yutaka Matsuno [*],
Yuichi Sei [†], Yasuyuki Tahara [†], Akihiko Ohsuga [†]

Abstract

Cyber-attacks targeting vulnerabilities in the internet of things (IoT) devices are increasing every year, and various methods and analysis tools for IoT vulnerability detection have been proposed. Concerning the analysis tools, reducing man-hours is crucial to ensure that users concentrate only on the work necessary to perform the analysis. However, the analysis tools proposed by existing studies are time-consuming in their setup, preprocessing, and analysis. To address these issues, we propose Yielding Optimal Device Analyzer (YODA) middleware for static analysis of IoT firmware. To define the requirements of YODA, we referred to the existing analysis tools, common middleware definitions, and typical middleware requirements for IoT environments. In the construction phase, we developed four functions and an innovation related to the setup. To evaluate the utility of YODA, we compared the time for setup, preprocessing, analysis and scalability with a baseline tool and Karonte, the most popular analysis tool available presently. Accordingly, we observed that YODA was faster than Karonte at every stage: the average setup time was 53 minutes, average preprocessing time was 2 hours and 31 minutes, and average analysis time was ˜11 hours for the analysis of a firmware with size 304 MB. Thus, we successfully developed the middleware that fulfills the requirements of firmware analysis.

Keywords: IoT, Middleware, Firmware Analysis, Static Analysis

1 Introduction

The internet of things (IoT) market is expected to expand at a compound annual growth rate of 21% from 2018 to 2025. Meanwhile, the attacks targeting IoT vulnerabilities have also increased from 231 million attacks on honeypots in the first half of 2018 to 4.2 billion in the second half of 2020, an 18-fold increase [1]. An example of IoT vulnerability is the existence of credentials that allow users to log into IoT servers in Zyxel routers. In this case, the username and password are embedded in the firmware of the router, allowing users to

[*] Nihon University, Chiba, Japan
[†] University of Electro-Communications, Tokyo, Japan

log in to the router. This case has been reported as a serious vulnerability in the common vulnerability and exposures (CVE) as CVE-2020-29583.

Various methods for detecting such IoT vulnerabilities have been proposed. One of the popular methods is static analysis. Static analysis detects vulnerabilities by analyzing programs without the execution of an IoT firmware. The flow of the analysis is as follows: The reverse-engineering, which extracts data, such as the source code, is applied to the binary firmware. Vulnerabilities are then detected by applying vulnerability detection algorithms to the reverse-engineering results. Karonte is the most popular static analysis method and has discovered a large number of vulnerabilities in recent years [2–6].

There are three issues common to existing analysis tools, including Karonte. The first issues is a redundant preprocessing. For example, to begin the analysis, the user must disassemble the firmware, extract the configuration files, and investigate the base address of the firmware. This preprocessing step is time-consuming owing to the complexity of the task.

The second issue is regarding the complexity in building the environment. We have confirmed that several existing analysis tools are unable to complete installation due to a lack of documentation and tool maintenance [7–10]. Among the existing analysis tools, only Karonte was successfully installed and executed on the sample analysis.

The complexity of setup is inconvenient as it hampers comprehensive vulnerability analysis. This is because each tool detects only a limited number of vulnerabilities, and thus, comprehensive vulnerability analysis requires multiple tools. For example, Karonte detects two types of vulnerabilities: buffer overflows and distributed denial of service (DDoS) attacks. Therefore, if a user wants to detect the highest vulnerability in the IoT vulnerability ranking—"embedding credentials"—the user must employ other analysis tools [11].

The last issue is the long time required for the analysis. In worst cases, analyzing a large size of firmware may require more than a day [2, 12]. Since a long analysis time leads to delays in vulnerability detection, reducing the analysis time is critical.

To address these issues, we propose Yielding Optimal Device Analyzer (YODA) middleware for static analysis of IoT firmware. Our middleware is available on Github[1]. In general, middleware is a software that abstracts the complexity of a system or hardware and provides services common to applications [13]. Herein, applications refer to static analysis tools. Through the use or development of applications that incorporate our middleware, users can reduce the time involved in the setup, preprocessing, and analysis, compared to those of Karonte. The setup phase represents the process beginning from installation to startup. The preprocessing phase represents the process beginning from execution up to reverse-engineering.

To the best of our knowledge, this is the first time a middleware is built for the static analysis of IoT firmwares. Therefore, we began by defining the requirements for our middleware tool. The requirements were defined based on the common definition of a middleware, a survey of requirements for middlewares in similar areas, and shortcomings of static analysis tools.

For evaluation, we surveyed 19 users to confirm that the middleware met the requirements. We compared the time for setup, preprocessing, and analysis for three tools: the proposed middleware, Karonte, and a baseline tool that provides the services necessary for the preprocessing; however, it does not consider the requirements of the middleware. The results indicated that our middleware required the least time and was the only tool for which

[1] `https://github.com/usaribbon/firmddle_docker`

all users completed all tasks within the allotted time.

In another experiment involving the analysis time and scalability, we compared the analysis time of Karonte with that of an example analysis tool developed using our middleware and containing the algorithm of Karonte. Consequently, we confirmed that our example analysis tool completed the analysis faster than Karonte for all 22 firmwares from four vendors. In the case with the largest difference, the analysis time was reduced by ˜11 hours.

Furthermore, scalability is crucial for middleware. Our middleware demonstrated superiority over Karonte in terms of both the number of incorporated vulnerability detection algorithms and the scope of analysis.

The structure of this paper is as follows: in Section 2, we summarize relevant research on firmware vulnerability analysis and related tools. Further, we discuss the results from our investigation of the analysis tools. In Section 3, we describe the survey conducted for studying the requirements for a middleware in IoT environments, which is an active area of middleware discussion and a field of study similar and relevant to our discussion; we subsequently define the requirements for our middleware. In Section 6, we discuss how the software design meets the defined requirements, including specific implementation methods. Section 5 describes the overall development of the middleware and the details of newly developed technologies. Section 7 presents the evaluation of the middleware. In Section 8, we discuss the evaluation and the level of completion of our middleware. Finally, in Section 9, we summarize this study and discuss future work. This paper synthesizes the findings presented in [14–16]. Furthermore, Section 3 delineates the recent updates implemented in the middleware.

2 Research on Firmware Vulnerability Analysis

We present 13 studies of firmware vulnerability analysis methods. We surveyed studies in which unknown vulnerabilities were discovered, including well-known studies with several citations and the most recent investigations. For studies utilizing publicly available analysis tools, we examined whether the analysis tools could be installed and executed on the samples that were provided in each case. If an analysis tool was distributed in a virtual environment such as Docker, we used the distributed environment. Otherwise, we installed the tool on a Docker container running Ubuntu 22.04. Table 1 summarizes the availability of the sample program of related research tools.

2.1 Summary

Among the 13 related studies, 5 studies published analysis tools and only 1 of them successfully executed the sample program according to the related official documentation. In fact, four tools could not be installed and did not execute the samples for the following reasons: the installation method was not described, the input data were unclear, and there were installation errors due to missing dependencies for the libraries.

The issues regarding a missing execution method and unclear input data can be fixed by improving the documentation. The library installation errors can be fixed by providing a virtual environment that includes the library.

For Karonte, which executed the sample program, we observed significant workload in startup and preprocessing. Karonte is a vulnerability detection system that focuses on

Table 1: Availability of executing sample program of related research tools

Research	Succed to Run Sample	Reason of Running Failure
FIRMADYNE [7]	-	Related library is not found
Stringer [8]	-	Unclear input data
HumIDIFy [9]	-	Non-description of usage
FirmFuz [10]z	-	Related library is not found
Firmalice [12], PIE [17], D-Taint [18], FIrmUp [19], John et al. [20], String Search [21], Socket Search [22], User Input Search [23, 24]	-	Tool is not publicly available
KARONTE [2]	✓	-
Number of ✓	1	-

inter-process communication, such as file sharing, shared memory architecture, environment variables, sockets, and command arguments [2]. The method monitors data communication between binaries by constructing graphs on a per-binary basis and correlates data communication and memory location with static taint analysis. Karonte is publicly available [25]. Particularly, Karonte's Docker container may require special options for startup; however, these options are not described in the documentation, causing unnecessary workload in the setup. Apart from the sample, firmware analysis required the creation of a configuration file as a preprocessing step, for which the man-hour workload was identified. In this case, the complexity of initiating the Docker container can be addressed by improving the documentation, and the complexity of preprocessing can be addressed by absorbing the content of the work into that performed by our middleware.

3 Requirements for Middleware of IoT and Define Requirements of Our Middleware

The middleware requirements for static analysis of IoT firmware have not been yet defined. A previous study proposed a concept of middleware for IoT firmware static analysis based on the common analysis function of other static analysis methods [26]. However, the definition of the middleware requirements for this field and development of the middleware is yet to be proposed. Therefore, first, this study defines the requirements of middleware. For proper definition of the requirements, we investigated the middleware requirements in the case of the IoT, which is a similar field with active middleware discussions. Based on our investigation, the common definition of middleware, and the issues of static analysis tools, we define the requirements that this middleware should have.

3.1 Middleware Requirements for IoT

The middleware requirements for the IoT environment have been discussed in a previous study [13, 27, 28]. Razzaque et al. surveyed 61 studies on the middleware of the field and consequently defined the requirements of a middleware [13]. Their exhaustive survey has been highly appreciated and is still referred to as a guideline for middleware development in IoT environments. However, their survey was focused on the research conducted before 2014. Thus, the recent middleware requirements with advances in the IoT environment must be examined.

Zhang et al. surveyed the middleware progress up to 2021 and then defined the middleware requirements for the recent IoT environment [27]. They set strict survey criteria for the selection of the studies, and ultimately 20 studies were chosen. Their criteria were that the study must be presented in peer-reviewed journals or articles indexed in Web of Science or International Scientific Indexing, which are recognized as a trusted citation databases. In another study, Vikash et al. summarized the requirements of middleware for wireless sensor networks, which is an essential technology for the realization of IoT environments [28].

These studies all reported that the important requirements of IoT middleware are ease of deployment, real-timeness, interoperability, light-weightness, scalability, security and privacy, and reliability and availability.

Ease of deployment implies that users can easily install and use the middleware without the requirement of expert knowledge or support. Real-timeness indicates the ability to quickly respond to a request via applications or users. Interoperability is the ability to provide services to various IoT devices. Light-weightness is the ability to run on various devices, thus reducing the use of computer resources and middleware size. Scalability is the ability to adapt to the scaling up of a system and the addition of new functions. Security and privacy involve the protection of personal information held by the middleware and ensuring user privacy; the disclosure of context-awareness of personal information is also important. Finally, reliability and availability are the ability to provide services even in the case of failures and minimize downtime.

3.2 Definition of Middleware Requirements for Static Analysis of IoT Firmware

We discuss about the definition of the middleware requirements for static analysis of IoT firmware. We refer to the survey of middleware requirements for IoT environments, the common middleware definitions, and the issue of existing static analysis tools to define the requirements. First, the common definition of middleware described in Section 1 states that the middleware must be usable without users being aware of the complexity of the middleware. Therefore, "Ease of deployment" is an important requirement for abstracting the complexity of setup and preprocessing, which are important issues in existing static analysis tools.

Next, the middleware must provide common services. At this point, services that solve the common problems of existing tools must be provided for the realization of better applications. Therefore, "Real-timeness" is an important requirement for solving the analysis time, which is a common problem of the existing tools.

Scalability is also an important factor. Middleware is software that runs between the application and the operating system, and its performance and scalability have a significant impact on the overall system.

Thus, the middleware requirements for static analysis of IoT firmware are summarized

as follows.

1. Ease of deployment
2. Real-timeness
3. Scalability

It is evident that the accuracy of vulnerability detection is an important requirement for static analysis tools. However, as detection accuracy depends on the application's vulnerability detection algorithms, it is outside the scope of a middleware requirement. Our middleware ensures that scalability, real-timeness, and ease of deployment are suitable to solve the current problem for existing tools.

4 Middleware Software Design

This section discusses the software design that satisfies the requirements for our middleware in 3.2.

4.1 Ease of Deployment

To ensure the ease of deployment, we proposed two features. The first feature involves allowing the firmware itself to be used as input data for the middleware. This function abstracts the preprocessing complexity, which is a problem encountered in existing tools. Thus, it reduces the time of preprocessing. This features is included in a component of "Gathering Firmware Information".

The second feature is "Middleware base" that is provided as a virtual environment. This promotes the realization of an easy setup of the middleware. It absorbs differences in software behavior and environment settings. We also distribute our middleware with sufficient documentation to make it easy to install and use the features.

4.2 Real-timeness

To ensure real-timeness, we proposed three features. The first feature is included in "Gathering Firmware Information" that is the prevention of duplicate analysis caused by symbolic links. Ghidra, which is the basis of reverse-engineering and analysis, might treat a symbolic link as an entity at the analysis stage, resulting in duplicate analysis and increasing the analysis time.

The second feature is referred to as "Analysis Base" that provides a batch reverse-engineering and applying detection algorithms a on multiple ELF files contained in the firmware. Ghidra necessitates the manual operation of the reverse-engineering and applying detection algorithms to all files, which is not realistic in terms of the man-hours required. Therefore, this study developed a technology capable of continuously applying reverse-engineering and detecting algorithms to multiple files.

The third feature is referred to as "Reverse-Enginnering" to provide reverse-engineering results as an API for each function. This facilitates algorithm development using only the necessary analysis results. Consequently, unnecessary analysis processing and time is reduced.

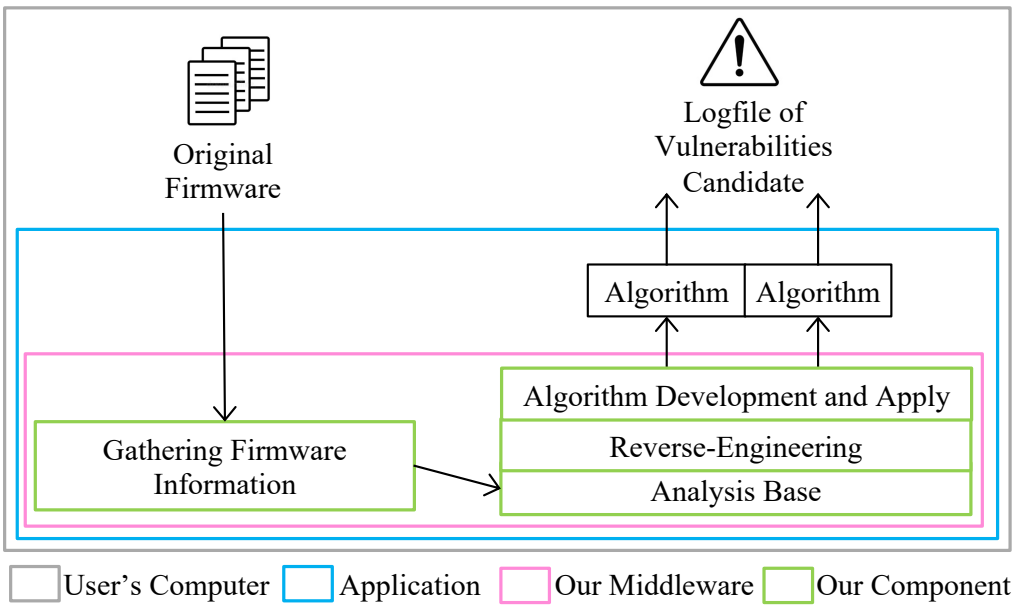

Figure 1: Overview of the proposed Yielding Optimal Device Analyzer(YODA) middleware

4.3 Scalability

To ensure scalability, it provides a base with a variety of analysis algorithms and functions to improve the coverage of the analysis range. The scalability of the middleware to integrate with various external functions is critical. One indicator of this scalability is the number of user-developed vulnerability detection algorithms that can be integrated into the middleware. Improving analysis coverage is also important to avoid analysis omissions. In this study, scalability is provided by the "Analysis Base".

4.4 Other Features for Common Middleware Requirements

We also provides an environment referred to as "Algorithm Development and Apply" for the development and applying of vulnerability detection algorithms to reverse-engineering results. This functionality is not part of the middleware requirements. However, as the role of middleware in common is to abstract complex processing and provide common services to applications, and owing to the essentiality of this feature for application development, this study incorporated it in our middleware.

5 Middleware Development

This section describes the middleware development in detail. Figure 1 presents an overview of our middleware.

5.1 Development of Middleware Base

The proposed middleware uses containerized virtualization technology based on Docker. It is a Linux container technology that packages all important files for software execution, including applications and runtime environments. Users can use the service by launching the Docker container of the middleware. This addresses the problem of complicated library installation required at setup, which has been a major problem for analysis tools in case of the related research described in Section 2.

5.2 Example of Analysis Tool using The Proposed Middleware

To present an example demonstrating the use of the proposed middleware, we built an analysis tool using the proposed middleware. This analysis tool contained a function similar to that as a part of the Karonte's vulnerability detection algorithm. To develop the analysis tool, the development function proposed in Section 6.3 was used.

6 Proposed Middleware

In this study, we propose and build a middleware for IoT static analysis that can be equipped with multiple vulnerability-detection algorithms and run in various environments. Through our middleware, multiple vulnerability-detection algorithms can be executed in a single analysis environment, which solves the scalability issue recurrent in existing research. In addition, we distributed our middleware in a virtual environment to run in various user environments. This solves the difficulty of tool set up. Our middleware is available with vulnerability-detection algorithms on Github. This section describes middleware development in detail. Figure 1 presents an overview of our middleware, which treats each service as a component. Figure 1 denotes the components with green squares.

6.1 Gathering Firmware Information

This component provides a service to automatically collect information from the firmware that is necessary for the analysis process. Before executing this component, the user stores the to-be-analyzed firmware binaries in a specified directory. The user can save multiple firmware binaries. Algorithm 1 depicts the program flow of this component. Upon executing this component, a compose file of the firmware is extracted at line 1. Because the method to extract files is well established, we did not develop a new extraction function but used the firmware-mod-kit [29].

Subsequently, the function originally developed in this study is executed between lines 2–13. First, a list of to-be-analyzed compose files is created and passed to the next process at line . In general, static analysis is performed on Executable and Linkable Format (ELF) files, thus, this component creates a list of ELF files. If other file formats are passed to the analysis function in the next step, the function is stopped.

In addition, the component eliminates duplicate files when creating the list of files. For example, the busybox command often included in firmware is called from many directories as symbolic links. If duplicates are not eliminated, busybox will be analyzed multiple times because some tools consider symbolic links as entities in the analysis. Therefore, duplicate analysis can affect analysis performance and duplicate files should be eliminated. If the file format is symbolic links, the file is ignored at line 7.

Algorithm 1 Gathering Firmware Information

```
1: FirmwareBinaryList ← getFirmwareBinaryList()
2: while FirmwareBinaryList.hasNext() do
3:     ELFList.open(ThisFirmwareLogFile)
4:     ExtractedFiles ← FirmwareBinaryList.getFirmwareFiles()
5:     while ExtractedFiles.hasNext() do
6:         File ← ExtractedFiles.getFile()
7:         if File.getFormat() = ELF then
8:             ELFList.write(File.getFilePath())
9:             File.ImporttoGhidra(this.projectPath)
10:        end if
11:    end while
12:    ELFList.close()
13: end while
```

6.2 Reverse Engineering

This component analyzes ELF files extracted from firmware. Analyzing ELF files is generally called reverse engineering, in which various information is obtained from the ELF file by restoring the original program, generating graphs, etc. The results obtained by reverse engineering can be used for advanced vulnerability analysis based on the flow of programs and data.

This component uses Ghidra as the core technology for reverse engineering. Ghidra, IDAPro, and angr are the most popular tools used as analysis bases [30]. IDAPro requires a paid license, which could deter users from employing our middleware. However, Ghidra uses Apache License 2.0, which is free to use. Although angr is free to use similar to Ghidra, the latter exhibits better ELF file analysis coverage compared to the former [31]. Thus, considering these factors, this study employed Ghidra.

In this component, we developed a function that allows Ghidra to automatically perform batch analysis based on the ELF file list. Generally, the number of ELF files included in a single firmware file ranges from tens to hundreds or more. However, Ghidra cannot automatically reverse engineer multiple ELF files, thus, this must be done manually.

6.3 Algorithm Development Base

This component provides an environment for developing vulnerability-detection algorithms using the results obtained in the previous step.

The component provides templates as Java files so that the users caneasily develop vulnerability-detection algorithms. The template file contains sample programs necessary for creating the bases of the algorithm, such as a program to obtain the reverse-engineering results and another to output the results via a log file. Utilizing the template files, users can easily access the services of the middleware and start developing essential algorithms.

As another unique technology, we have implemented a function that provides reverse-engineering results as an API, allowing users to easily obtain analysis results with a few lines of code. Normally, the user would have to write an extensive amount of code to obtain the reverse-engineering results. Moreover, Ghidra's official guidelines do not describe how to obtain the results. In other words, it is necessary to decipher Ghidra's internal programs, which requires complicated program development. Through this API, users do not need

Algorithm 2 setUpDecompiler()

```
1: decompApi ← FlatDecompilerAPI(firmware)
2: if decompApi.getDecompiler() == null then
3:     decompApi.initialize()
4: end if
5: decomplib ← decompApi.getDecompiler()
6: options = DecompileOptions()
7: decomplib.toggleCCode(true)
8: decomplib.toggleSyntaxTree(true)
9: decomplib.setSimplificationStyle("decompile")
10: returndecompApi;
```

to develop complicated programs separately and can use the reverse-engineering results returned by the corresponding detection algorithms simply by calling the API.

We present an example of calling the API at Algorithm 3. This API decompiles firmware binary and returns the original program of firmware. Decompiled firmware program contains important information that is used to analyze the firmware. However, this API is not contained in official Ghidra's document. We created this API to realize an ease of development.

6.4 Example of Analysis Tool using The Proposed Middleware

As an example demonstrating the use of the proposed middleware, we built an analysis tool running the proposed middleware. This analysis tool contained a function similar to that used as a part of the Karonte's vulnerability detection algorithm.

6.4.1 Network Communication Detection Algorithm

This algorithm forms a part of the vulnerability detection functionality of Karonte [2]. It is used to ensure that ELF files containing strings related to network communication accept DDoS attack inputs from cyber-attackers. Therefore, it detects ELF files containing these strings as follows `QUERY_STRING`, `username`,`http_`,`REMOTE_ADDR`,`boundary=`,`HTTP_`,`query`, `_remote`,`user-agent`,`soap`,`index`,
`CONTENT_TYPE`,`Content-Type`.

Algorithm 3 Example of analysis tool

```
1: logPath ← "/mnt/example/logs"
2: stringsSearch ← StringSearchHeadless()
3: stringsSearch.getVals(targetFirmware, logPath)
```

6.4.2 Hard Coded Detection Algorithm

We constructed an example analysis tool that detects an embedded login information with our middleware. We demonstrated the tool by using the String Search, Socket Search, and User Input Search algorithms [21–23].

String Search is an algorithm that targets detecting embedded login information. It locates a function by searching for codes that use the strcmp() or strncmp() symbols. These

symbols are used to compare user input with embedded login information. Thus, this method categorizes functions that use these symbols as backdoor function candidates.

Socket Search locates codes that use the strcmp() or strncmp() symbols around a socket function. According to vulnerability-finding research, a backdoor is always accessible through the TCP/UDP function. Thus, the hardcoded strings around the socket symbols are denoted as candidates for login information.

User Input Search is a method for detecting embedded login information (username and password) in IoT devices using static analysis focusing on the user input value, as users must enter correct login values to access IoT devices.

We developed these algorithms using an Algorithm Development Base template. All algorithms are available on Github. Algorithm 3 presents the String Search algorithm constructed using our template. Users need to define a log file path for recording results at line 1 and run this program; thereafter the algorithm starts the analysis.

7 Evaluation

This section evaluates whether the proposed middleware satisfies these requirements.

Table 2: Comparison of Setup and Preprocessing Time

Tool Name	Setup (hour:min)			Preprocessing (hour:min)		
	Minimum	Maximum	Average	Minimum	Maximum	Average
Our Middleware	0:27	3:33	1:53	0:29	2:09	1:04
Karonte	0:12	4:55	2:46	1:14	6:00*	3:36*
Baseline	0:22	1:15	0:43	0:46	7:32*	3:30*

* The total time for setup and preprocessing was limited to 8 h; without the 8 h time limit, these work times would have increased.

Table 3: Comparison of Users

Tool Name	Number of user person	Number of users who completed all tasks in timeperson
Our Middleware	6	6
Karonte	7	2
Baseline	6	5

7.1 Ease of Deployment

To evaluate the ease of deployment, that is, low number of man-hours required to install the middleware and use the service, the setup and preprocessing time were measured. Here,

Table 4: Comparison of Total Time

Tool Name	Minimum	Maximum	Average
Our Middleware	0:56	5:42	2:58
Karonte	2:02	8:00*	6:22*
Baseline	1:08	8:00*	4:13*

* If the work was not completed in 8 h, it was considered to be 8 h. If the 8 h time limit is not set, it would increase.

three tools were evaluated: the proposed middleware, Karonte, and the baseline. The baseline is an environment that provides the services necessary for the preprocessing of our middleware; however, it does not consider the requirements. In the experiments, the subjects were provided with documentation for the tools and were asked about the time required to complete each task. The maximum working time was 8 h. The number of subjects employed for each tool was 6, 7, and 6 for the middleware, Karonte, and baseline, respectively.

The subjects were undergraduate and graduate students (Master and Doctoral Courses) in the field of computer science and software engineers, with at least five years of software development experience. Author's affiliation approved this experiment (Management ID: 22090), and prior written consent was obtained from each participant. Further, the native language of all subjects is Japanese, we translated the documentation of Karonte into Japanese to ensure fairness in the time required to decipher the documentation. Additionally, the number of subjects for Karonte was different from that for the other two tools because of the difference in the duration of the experiment.

Table 2 and 3 presents the results for each work time. In the setup phase, the proposed middleware completed the work, on average, 53 min earlier than Karonte, a maximum of 1 h and 22 min earlier. The minimum outcome was 15 min faster for Karonte. The baseline was completed earlier than our middleware.

During the preprocessing phase, our middleware completed the work faster than Karonte by an average of 2 h and 25 min, a maximum of 3 h and 11 min, and a minimum of 45 min. The middleware reduced the work time compared to the baseline by an average of 2 h 19 min, a maximum of 4 h 43 min, and a minimum of 17 min. All six subjects completed the entire task within 8 h in the case of the middleware. However, two of seven and five of six could not complete it within 8 h in the case of Karonte and baseline, respectively.

Table 4 shows the total time results. The average, maximum, and minimum total times for the setup and preprocessing tasks were 3 h and 18 min, 1 h and 38 min, and 1 h and 6 m, respectively, faster than Karonte. Compared to the baseline, the middleware completed the work 1 h and 9 min earlier on average, 1 h and 38 min earlier in case of the maximum time difference, and 12 min earlier in case of the minimum time difference.

7.2 Real-timeness

We compared the analysis time of Karonte with the example analysis tool developed in Section 6.4.1. Here, Karonte's dataset of 22 firmware from 4 vendors was used as the analysis target. For a fair comparison, we modified the analysis range of the Karonte program so that the detection ranges of Karonte and the sample tools were comparable. The computer

used to measure the analysis time had an Ubuntu 22.04.02 operating system, an i9-9900K CPU, and 16 GB of RAM.

Table 6 presents the analysis time results. For the largest firmware file (304 MB), the analysis using the sample tool was completed 10 h and 55 min earlier than that using Karonte. For TP-Link's 28 MB firmware, which has the smallest file size, the analysis was completed 14 min earlier than Karonte when using the sample tool.

Table 5: Analysis Time for Karonte and Our middleware

Vendor	Firmware Name	Unpacked Firmware Size(MB)	Our Middleware (hour:min)	Karonte (hour:min)
NETGEAR	R8500	304	0:37	11:32
NETGEAR	AC1450	71	0:13	0:40
TP_Link	Archer_C2	54	0:08	0:56
TP_Link	Archer_C50	53	0:08	0:55
TP_Link	Archer_C3200	96	0:13	1:31
TP_Link	Archer_D2	100	0:09	1:07
TP_Link	TD_W9970	57	0:08	0:54
TP_Link	TL-MR3020	57	0:04	0:18
TP_Link	TL-WA830RE	28	0:04	0:18
TP_Link	TL-WR1043ND	39	0:06	0:49
TP_Link	TX-VG1530	106	0:19	1:24
D-Link	DIR-868	71	0:11	8:52
D-Link	DIR-880	86	0:20	0:56
D-Link	DIR-885	104	0:12	0:42
D-Link	DIR-895	192	0:12	0:41
D-Link	DIR-118	40	0:10	0:40
D-Link	DIR-826	46	0:10	1:15
D-Link	DIR-890	105	0:13	0:43
Tenda	US_AC6V	33	0:07	1:13
Tenda	US_AC9V	37	0:09	2:07
Tenda	US_AC15V	64	0:12	2:55
Tenda	US_WH450AV	37	0:15	1:39

Table 7 reports the comparison results and confirms that our middleware can install five algorithms, whereas Karonte can run only two.

7.3 Scalability

7.3.1 Analysis Coverage

To investigate the performance of our middleware, we compared Karonte's analysis coverage with that of the analysis tool developed in Section 6.4. We analyzed 16 real-world firmware taken from three vendors. In the experimental procedure, we first run each tool on the same computer and analyze all the firmware. After the analysis, we manually checked the number of analyzed firmware files from the log file, and the analysis coverage is measured by comparing the number of analyzed files.

Table 6: Number of Analyzed ELF files

Vendor	Firmware Name	Proposed middleware	Karonte
Tenda	FH-1201	103	62
Tenda	FH-1206	103	62
Tenda	WH-450	149	134
TP_Link	Archer_C50	123	88
TP_Link	Archer_C2	124	89
TP_Link	Archer_C20	136	89
TP_Link	TD-W8970	175	108
TP_Link	TD-W9970	150	95
TP_Link	TL-MR3020	165	77
TP_Link	TL-MR3040	165	77
TP_Link	TL-WA701ND	153	72
TP_Link	TL-WA830RE	145	72
TP_Link	TL-WR1043ND	190	84
D-Link	DWR-118	226	180
D-Link	DIR-826	128	101
D-Link	DIR-842	91	79

Table 6 lists the number of ELF files analyzed by our middleware and Karonte. The results demonstrate that the number of files analyzed by our middleware is greater than those analyzed by Karonte for all 16 firmware. When evaluating scalability, we compared the number of vulnerability detection algorithms that can be implemented in our middleware to those compatible with Karonte.

7.3.2 Algorithm Scalability

We also counted the number of vulnerability-detection algorithms implemented on our middleware and Karonte. The first is a memory-corruption detection algorithm, which detects a vulnerability that corrupts data through unauthorized memory rewriting and manipulation. This is currently one of the most important vulnerabilities, as it affects not only IoT but also general-purpose computers. Secondly, we implemented an algorithm capable of detecting DoS vulnerabilities.This vulnerability is also the cause of the large-scale Mirai attack, which exploited vulnerabilities in IoT devices, and an important target for detection. The third is an embedded login information, which is ranked #1 in OWASP's IoT vulnerability ranking. This vulnerability is an important target for detection because IDs and passwords, which are login information, are written directly into the firmware, allowing any third party who knows the login information to log into the firmware. Table 7 reports the comparison results and confirms that our middleware can install five algorithms, whereas Karonte can run only two.

Table 7: Comparison of available on-board algorithms

Vulnerability	Our Middleware	Karonte
Memory-corruption detection	✓	✓
DoS vulnerabilities detection	✓	✓
Embedded Login Information (String Search, Socket Search, User Input Search)	✓	-

8 Discussion

8.1 Ease of Deployment

During the setup phase, we confirmed that the average and maximum time required for our middleware was shorter than that for Karonte. Karonte was faster in the case of the minimum value because of the difference in the number of external tools installed or a startup confirmation: Karonte only required the installation of Docker, while the middleware required the installation of Docker and Git commands. Therefore, the minimum value for the proposed middleware was longer than that of Karonte owing to the installation time of the Git command incorporated in our middleware. However, Docker and Git are common technologies, and certain subjects use this software regularly and may have already installed them before performing the experiment. Therefore, the difference in time owing to the installation or confirmation of external tools is not unique to the analysis tools.

The baseline was completed in less time than the proposed middleware. This is because the baseline only required Ghidra installation. Our middleware and Karonte necessitated multiple steps to download, install, and verify the tool startup, which increased the work time. However, our middleware base and Karonte are packaged with the necessary tools included after preprocessing; thus, no additional installation work was required. As the scope of this experiment was to install the tools necessary for preprocessing, if all the tools were to be installed following preprocessing beyond the baseline, the setup time would increase.

In terms of total time, our middleware was found to require less time for all values compared to Karonte and baseline. Among the three tools, the subjects could complete all their work within 8 h only for our middleware. However, one of six subjects in the baseline and five of seven subjects in Karonte could not complete the entire process within the time limit. Therefore, if no upper time limits were set for this experiment, the working time for baseline and Karonte would increase. This result indicates that our middleware successfully reduced user workload.

Next, we discuss the subject's reports regarding the performance of all tasks. According to the reports of the subjects who used our middleware, certain subjects required a certain time to confirm the use of Docker during the setup phase. However, no errors related to the installation of the middleware were reported. Similarly, no errors were reported in the preprocessing work, and all subjects could complete the work within the required time. However, we received one report that an operation was unclear, although this was

not an error. The report said that they were confused regarding the environment to be used to execute the services because our middleware provides a graphical user interface and a command-line interface (CLI) environment. The design allows a user to use either of these environments to perform services. However, to further simplify the implementation, we could improve our documentation to state that the service can be used in either environment, or that the CLI environment is the recommended environment.

According to the work reports of the subjects who used Karonte, several of them required a long time to start the Docker container of Karonte because it required special options instead of the usual Docker commands, which were not described in their documentation. In contrast, the subjects using our middleware did not report any errors or unclear points during startup. Thus, the problems that occurred with Karonte can be solved using the proposed middleware. In addition, certain subjects had trouble understanding the installation and use of Docker. Thus, the proposed middleware was the only one among the three tools using which all subjects completed their tasks in time, and all tasks were completed in a shorter time than Karonte. The subjects of the proposed middleware did not report any complexity of preprocessing tasks that occurred with Karonte or the baseline. In addition, no interruptions owing to errors related to installation and use caused by inadequate documentation were reported. These results indicate that the proposed middleware is easy to install and use and satisfies the requirements of ease of deployment.

8.2 Real-timeness

In all the 22 firmware cases, the analysis using the middleware was faster than that using Karonte. In particular, the analysis time varied with the increase of firmware size, with the largest firmware, NETGEAR's 304 MB firmware, exhibiting a difference of 10 h and 55 min. The reason for the shorter analysis time using the proposed middleware is that it allows the algorithm to select the reverse-engineering results by API at Reverse-Engineering function. This reduces unnecessary processing and significantly shortens the analysis time. In contrast, Karonte includes a result of a flow graph of the ELF file as a basic reverse-engineering result, which is not required by the detection algorithm. The graph construction is such that the greater the number of functions, the greater is the number of function connections and time required to complete the process.

The reason the baseline was not included in this evaluation was that the baseline included work in addition to the analysis time, because the majority of the work was manual in nature, rendering accurate measurements difficult. Even if a documentation man-hour problem of the baseline is solved and analysis becomes possible, duplicate analysis would still occur in the baseline. Therefore, it is expected that the baseline will require more analysis time than our middleware, which eliminates the time of duplicate analysis by function of Gathering Firmware Information. Thus, the proposed middleware satisfies the requirement of real-timeness, i.e., it exhibits the ability to perform the required processing quickly.

In these experiments, we demonstrated that the proposed middleware enhances the analysis speed while preserving the vulnerability detection performance of Karonte. Furthermore, the proposed middleware is anticipated to improve the analysis speed of tools beyond Karonte as well. This is attributed to our API, which selectively extracts critical reverse-engineering results for the algorithm, and a feature that eliminates redundant analysis, both of which contribute to the improvement of analysis time for various tools. The implementation and evaluation of diverse analysis tools to assess execution speed remains a future endeavor.

8.3 Scalability

8.3.1 Analisys Coverage

The number of analyzed files was higher for our middleware than for Karonte as the methods for collecting ELF files differed between the two tools. Karonte uses the Linux find command to retrieve ELF files from firmware. However, this was not effective in retrieving some of the Linux Kernel Module (.ko) and shared library (.so) files. In addition, image files (.png, .bmp) and text files (.xml, .js, .conf), which are not among the analysis targets, were also acquired, resulting in many false positives. For example, the D-Link product provides a management system that operates via a web browser, for which the programs and images that make up the management screen were collected. False positives should not be generated because such files, i.e., files that are not subject to analysis, may cause a read error in the next reverse-engineering phase and stop the analysis.

However, our middleware assesses the target files one by one and extracts only those identified as ELF files. Therefore, our middleware was able to extract ELF files exhaustively and perform the analysis without generating false positives. The above results underline that the our middleware improves the analysis coverage and confirm its usefulness.

8.3.2 Algorithm Scalability

As a result, our middleware can implement more vulnerability-detection algorithms than those compatible with Karonte, confirming the superiority of our middleware's scalability. The reason Karonte cannot support an embedded login information detection algorithm is that the function required for such detection is not implemented. To detect this vulnerability, a decompiled binary program must be parsed for password strings. However, Karonte's program was commented out, rendering it unavailable. We tried to run the commented-out program, an error occurred and program could not be executed. For these reasons, embedded login information detection cannot be implemented in Karonte. However, our middleware provides an API for string analysis of decompiled programs, which enables the implementation of an embedded login information detection algorithm.

Karonte presents challenges in terms of incorporating algorithms as well as extending its functionality owing to the complexities of the firmware input and output programs. For example, there were several programs that output the results of applying the algorithm; however, it was unclear which one to call to output the results of a specific result file. The complexity of these programs makes it difficult to add vulnerability-detection algorithms to the system, which would require additional development effort.

9 Conclusion

This study built and evaluated a middleware to perform static analysis of IoT firmware to address the problems of existing tools, such as the setup complexity, preprocessing, and long analysis time. First, the requirements were defined to build the middleware. When defining the requirements, the problems of existing analysis tools, common middleware definitions, and middleware requirements for IoT environments in similar areas where middleware proposals are popular were investigated. During the development of the middleware, we introduced four new functions and one environment construction innovation to realize a middleware that satisfied the requirements. For the evaluation of the proposed middleware, we compared the setup, preprocessing, and analysis time with the baseline

and Karonte, which are the most popular middleware among the existing studies, to assess whether the proposed middleware satisfied the requirements. Furthermore, to evaluate the scalability of the middleware, we conducted an investigation into the analytical coverage and the number of implemented algorithms. Consequently, we confirmed that our middleware outperformed Karonte in all work items, and we succeeded in developing a middleware that satisfied the requirements of this study.

In the future, we will further expand the capabilities of the middleware and promote its use by improving the API for reverse-engineering results. We will also develop more example analysis tools that incorporate the algorithms of other existing tools and evaluate an analysis time. In addition, we will continue to keep up with analysis tools proposed in the future and consider requirements and additional features for problems that can be solved using our proposed middleware.

Although this study has used analysis scope (i.e., number of ELF files) and algorithm count as metrics, measuring actual vulnerability detection counts, false positive and false negative rates, and evaluating detection accuracy against established benchmarks (e.g., CVE repositories) would more rigorously demonstrate YODA's practical utility.

References

[1] M. Alsheikh, L. Konieczny, M. Prater, G. Smith, and S. Uludag, "The state of iot security: Unequivocal appeal to cybercriminals, onerous to defenders," *IEEE Consumer Electronics Magazine*, vol. 11, no. 3, pp. 59–68, 2022.

[2] N. Redini, A. MacHiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Karonte: Detecting insecure multi-binary interactions in embedded firmware," in *Proc. 2020 IEEE Symposium on Security and Privacy*, May. 2020.

[3] J. Yun, F. Rustamov, J. Kim, and Y. Shin, "Fuzzing of embedded systems: A survey," *ACM Comput. Surv.*, vol. 55, no. 7, pp. 1–33, 2022.

[4] X. Feng, X. Zhu, Q.-L. Han, W. Zhou, S. Wen, and Y. Xiang, "Detecting vulnerability on iot device firmware: A survey," *IEEE/CAA Journal of Automatica Sinica*, vol. 10, no. 1, pp. 25–41, 2023.

[5] Z. Gao, C. Zhang, H. Liu, W. Sun, Z. Tang, L. Jiang, J. Chen, and Y. Xie, "Faster and better: Detecting vulnerabilities in linux-based iot firmware with optimized reaching definition analysis," *Proceedings 2024 Network and Distributed System Security Symposium*, Feb. 2024.

[6] T. Bakhshi, B. Ghita, and I. Kuzminykh, "A review of iot firmware vulnerabilities and auditing techniques," *Sensors*, vol. 24, no. 2, 2024.

[7] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware," in *Proc. 23rd Annual Network and Distributed System Security Symposium*, (San Diego, USA), Feb. 2016.

[8] S. L. Thomas, T. Chothia, and F. D. Garcia, "Stringer: Measuring the importance of static data comparisons to detect backdoors and undocumented functionality," in

Proc. 22nd European Symposium on Research in Computer Security, (Copenhagen, Denmark), pp. 513–531, Sept. 2017.

[9] S. L. Thomas, T. Chothia, and F. D. Garcia, "Humidify: A tool for hidden functionality detection in firmware," in *Proc. 24rd Annual Network and Distributed System Security Symposium*, (San Diego, USA), pp. 279–300, Feb. 2017.

[10] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. Shrobe, and M. Payer, "Firmfuzz: Automated iot firmware introspection and analysis," in *Proc. the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*, (London, United Kingdom), p. 15–21, Nov. 2019.

[11] P. Ferrara, A. K. Mandal, A. Cortesi, and F. Spoto, "Static analysis for discovering iot vulnerabilities," *Int. J. Softw. Tools Technol. Transf.*, vol. 23, no. 1, p. 71–88, 2021.

[12] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware.," in *Proc. 22rd Annual Network and Distributed System Security Symposium*, (San Diego, USA), Feb. 2015.

[13] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke, "Middleware for internet of things: A survey," *IEEE Internet of Things Journal*, vol. 3, no. 1, pp. 70–95, 2016.

[14] M. Yoda, S. Nakamura, Y. Sei, Y. Tahara, and A. Ohsuga, "A middleware to improve analysis coverage in iot vulnerability detection," in *Proc. 26th IEEE/ACIS International Winter Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, (Taichung, Taiwan), Dec. 2023.

[15] M. Yoda, S. Nakamura, Y. Sei, Y. Tahara, and A. Ohsuga, "A middleware to improve analysis coverage in iot vulnerability detection," in *IEEE International Conference on Internet of Things and Intelligence Systems, IoTaIS 2023, Bali, Indonesia, November 28-30, 2023*, pp. 103–107, IEEE, 2023.

[16] M. Yoda, S. Nakamura, Y. Sei, Y. Matsuno, Y. Tahara, and A. Ohsuga, "Yoda: Middleware of static analysis tools for iot firmware - proposal and evaluation of time reducing mechanisms for setup, preprocessing, and analysis," in *Proc. 18th International Conference on E-Service and Knowledge Management*, (Kagawa, Japan), Jul. 2024.

[17] L. Cojocar, J. Zaddach, R. Verdult, H. Bos, A. Francillon, and D. Balzarotti, "PIE: parser identification in embedded systems," in *Proc. the 31st Annual Computer Security Applications Conference*, (Los Angeles, USA), pp. 251–260, Dec. 2015.

[18] K. Cheng, Q. Li, L. Wang, Q. Chen, Y. Zheng, L. Sun, and Z. Liang, "Dtaint: Detecting the taint-style vulnerability in embedded device firmware," (Luxembourg, Luxembourg), pp. 430–441, 2018.

[19] Y. David, N. Partush, and E. Yahav, "Firmup: Precise static detection of common vulnerabilities in firmware," in *Proc. the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, (New York, USA), p. 392–404, Mar. 2018.

[20] T. S. John, T. Thomas, and S. Emmanuel, "Graph convolutional networks for android malware detection with system call graphs," in *Proc. Third ISEA Conference on Security and Privacy*, (Guwahati, India), pp. 162–170, Feb. 2020.

[21] M. Yoda, S. Sakuraba, Y. Sei, Y. Tahara, and A. Ohsuga, "Detection of the hardcoded login information from socket and string compare symbols," *2021 Annals of Emerging Technologies in Computing*, vol. 5, no. 1, pp. 28–39, 2021.

[22] M. Yoda, S. Sakuraba, Y. Sei, Y. Tahara, and A. Ohsuga, "Detection of the hardcoded login information from socket symbols," in *Proc. 3rd IEEE International Conference on Computing, Electronics & Communications Engineering*, (Essex, United Kingdom), pp. 33–38, Aug. 2020.

[23] M. Yoda, S. Sakuraba, Y. Sei, Y. Tahara, and A. Ohsuga, "Detecting hardcoded login information from user input," in *Proc. IEEE 41st International Conference on Consumer Electronics*, pp. 104–105, Oct. 2022.

[24] M. Yoda, S. Sakuraba, Y. Sei, Y. Tahara, and A. Ohsuga, "Detection of plaintext login information in firmware," in *Proc. 2022 IEEE International Conference on Consumer Electronics – Taiwan*, (Taipei, Taiwan), pp. 1–2, Jul. 2022.

[25] N. Redini, A. MacHiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Karonte." `https://hub.docker.com/r/badnack/karonte`. accessed on May. 05. 2023.

[26] M. Yoda, S. Sakuraba, Y. Sei, Y. Tahara, and A. Ohsuga, "Proposal of a middleware to support development of iot firmware analysis tools," in *Proc. the 14th International Joint Conference on Knowledge-Based Software Engineering*, (Larnaca, Cyprus), pp. 3–14, Aug. 2023.

[27] J. Zhang, M. Ma, P. Wang, and X. dong Sun, "Middleware for the internet of things: A survey on requirements, enabling technologies, and solutions," *Journal of Systems Architecture*, vol. 117, p. 102098, 2021.

[28] Vikash, L. Mishra, and S. Varma, "Middleware technologies for smart wireless sensor networks towards internet of things: A comparative review," *Wireless Personal Communications*, vol. 116, pp. 1539–1574, Feb. 2021.

[29] J. Collake, "Firmware mod kit." `https://github.com/amitv87/firmware-mod-kit`. accessed on May. 05. 2023.

[30] Z. Zhang, Y. Ye, W. You, G. Tao, W.-c. Lee, Y. Kwon, Y. Aafer, and X. Zhang, "Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary," in *Proc. the 42nd IEEE Symposium on Security and Privacy*, (San Francisco, USA), pp. 813–832, May. 2021.

[31] C. Pang, R. Yu, D. Xu, E. Koskinen, G. Portokalidis, and J. Xu, "Towards optimal use of exception handling information for function detection," in *Proc. 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, (Online), pp. 338–349, 2021.