2021 Ninth International Symposium on Computing and Networking Workshops (CANDARW) | 978-1-6654-2835-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/CANDARW53999.2021.00074

FirmPot: A Framework for Intelligent-Interaction Honeypots Using Firmware of IoT Devices

Moeka Yamamoto, Shohei Kakei, and Shoichi Saito
Nagoya Institute of Technology
Aichi, Japan
Email: ckv14138@nitech.jp, {kakei.shohei, shoichi}@nitech.ac.jp

***Abstract*—IoT honeypots that mimic the behavior of IoT devices for threat analysis are becoming increasingly important. Existing honeypot systems use devices with a specific version of firmware installed to monitor cyber attacks. However, honeypots frequently receive requests targeting devices and firmware that are different from themselves. When honeypots return an error response to such a request, the attack is terminated, and the monitoring fails.**

To solve this problem, we introduce FirmPot, a framework that automatically generates intelligent-interaction honeypots using firmware. This framework has a firmware emulator optimized for honeypot generation and learns the behavior of embedded applications by using machine learning. The generated honeypots continue to interact with attackers by a mechanism that returns the best from the emulated responses to the attack request instead of an error response.

We experimented on embedded web applications of wireless routers based on the open-source OpenWrt. As a result, our framework generated honeypots that mimicked the embedded web applications of eight vendors and ten different CPU architectures. Furthermore, our approach to the interaction improved the session length with attackers compared to existing ones.

***Index Terms*—IoT Honeypot, Firmware, Machine Learning**

I. Introduction

IoT devices are making significant progress in increasing functionality. For example, the firmware of IoT devices often embeds web applications that allow users to check the device's status from the Internet [1]. Embedded web applications have a vendor-specific implementation. Thus, it is challenging to discover their vulnerabilities by static or dynamic analysis, and they are used without any security measure [2].

To solve the security problems of IoT devices, researchers focus on analyzing the attack methods using honeypots. Honeypots monitor an attacker's intrusion by pretending to be a legitimate device. If an attacker exploits a zero-day vulnerability, researchers can disseminate the information for developers and users of IoT devices.

Existing honeypot systems use IoT devices with a specific version of the firmware installed. However, the existence of vulnerabilities depends on the firmware version. The reason is that firmware updates partially modify the implementation. The modifying in implementation may remove existing vulnerabilities or may introduce new ones. Knowing this characteristic, attackers will check the device's firmware version before executing the exploit code. Once the attacker confirms that the honeypot is not the target system, the attack is terminated.

Our goal is to continue interacting with attackers to observe attacks targeting IoT devices effectively. In order to achieve our goal, honeypots must deal with system investigations by attackers. However, some system checks are simple, such as verify that the target system's response is not an error. Thus, a honeypot that returns a proper response instead of an error to the received request may bypass their checks.

We propose FirmPot, a framework that generates intelligent-interaction honeypots by emulating firmware for monitoring attacks targeting IoT devices. FirmPot collects web interactions by emulating firmware launched on a docker container and learns the correspondence of requests and responses by machine learning. Furthermore, the generated honeypots return the best from the emulated responses to the received request. With this mechanism, our honeypots do not return the error response even if the request is not collected at the emulating phase, increasing the likelihood that the attacks will continue.

We implemented the framework and experimented on wireless routers based on the open-source OpenWrt [3]. The results show that the framework can automatically generate honeypots that mimic the web applications of eight vendors and ten CPU architectures. Additionally, the honeypots generated by the framework were placed on the Internet and captured manipulation of the interface by attackers, such as login attempts. Furthermore, our approach to the interaction improved the session length with attackers compared to the existing ones.

The contributions of this study are as follows:

1. We have found an OpenWrt-based firmware emulation scheme that is suitable for honeypot generation.
2. We proposed an effective interaction method with attackers using machine learning.
3. We have established a framework to automatically generate honeypots by simply submitting firmware and release the source code to support future research[1].

The rest of the paper is structured as follows: Section 2 presents related works on IoT honeypots, and Section 3 provides the research background. Next, we present our framework in Section 4. In Section 5, we report our evaluation results, and in Section 6, we discuss our results and future work. Section 7 explains ethical considerations, and Section 8 concludes this paper.

[1]https://github.com/SaitoLab-Nitech/FirmPot

II. Related Work

IoT honeypots are a technology for threat analysis of IoT devices. They are mainly classified into two categories based on the level of interaction: high-interaction honeypots and low-interaction honeypots. In addition to these two categories, this section describes another interaction level, intelligent-interaction honeypots.

A. High-Interaction Honeypot

High-interaction honeypots collect advanced information of cyber attacks by providing systems in which attackers have complete control. IoT honeypots in this category include SIPHON [4], which deploys a physical device as a honeypot, and Honware [5], which exposes a firmware image running in a virtual environment.

The problem with high-interaction is that as the number of honeypots increases, scalability decreases. The reason is that physical devices and virtual machines consume computational resources. Furthermore, high-interaction honeypots are at risk of vulnerability exploitation, like Honware caught in a DDoS attack [5]. To prevent the exploitation of honeypots, users must regularly perform burdensome tasks such as state monitoring and system refresh.

B. Low-Interaction Honeypot

Low-interaction honeypots include those that emulate a single protocol, such as U-PoT [6], and those that emulate a specific device, such as ThingPot [7]. On the other hand, some honeypots scan the Internet to emulate multiple protocols of various IoT devices [8], [9].

Low-interaction honeypots support only some functions of the system, not the entire system. For example, few honeypots have the authentication functions provided by IoT devices. This problem stems from the lack of a generic emulation method suitable for IoT devices with different implementations by device types and vendors. In addition, it is easy for attackers to detect honeypots because of their fixed behavior when receiving a non-emulated request.

C. Intelligent-Interaction Honeypot

Intelligent-interaction honeypots have been added to the level of interaction by IoTCandyJar, the work of Luo et al. [8]. The concept of intelligent-interaction honeypots is to interact with attackers that maximize the likelihood of catching the attacks instead of accurately emulating the behavior of a specific device as in high-interaction or low-interaction.

IoTCandyJar selects what attackers expect from the many responses of IoT devices collected by internet scanning. If the expected response is selected, the attackers assume that the honeypot is their target and send a malicious command. IoTCandyJar uses a Markov decision model to learn from scratch what response an attacker expects. Thus, their honeypots take some time until they can respond appropriately to requests. As a result, it took two weeks for the model to learn enough to interact with attackers continuously.

III. Background and Motivation

This section explains the challenges in our framework.

A. Challenge for Firmware Emulation

The first step in our framework is to emulate the firmware of IoT devices. Our target firmware consists of a Linux-based kernel and filesystem. The applications stored in the filesystem are characterized by access to hardware. For this reason, QEMU is often used for emulating CPU and peripherals.

There are two approaches to QEMU: full-system emulation and user-mode emulation. Most existing firmware emulators use full-system emulation, which emulates the entire system, including the kernel. A typical emulator, Firmadyne [10], launches each application of the firmware through the boot process. However, Firmadyne has a low success rate of emulation, while Honware [5] and FirmAE [11] use an empirical approach to improving it.

These emulators commonly target MIPS big-endian, MIPS little-endian, and ARM little-endian. However, we found PowerPC and MIPS64 firmware images, as shown in Section V-A, so these emulators are less generic. In addition, Zheng et al. found that full-system emulation was approximately ten times slower than user-mode one, emulating only the target program [12]. However, user-mode emulation does not have access to kernel functions. Thus, our first challenge is to find the best emulation for speed and functionality in honeypot generation.

B. Challenge for Web Emulation

An embedded web application is a service for managing devices. It provides both static contents, such as HTML, and dynamic contents generated by scripts. Thus, attackers consider a web service with only static contents as a decoy system. In order to reproduce the essence of the web, Musch et al. [13] collect static contents by crawling and dynamic contents by fuzzing, which intentionally changes the response. However, they have only mimicked the pre-login pages. Most attacks against embedded web applications target their configuration pages that are accessible after login. Thus, honeypots without login-based interfaces are limited in the observed attacks. From the above, our second challenge is to mimic all pages and functions of an embedded web application.

C. Challenge for Intelligent Interaction

Many of today's attacks are executed by automated scripts. The scripts have pre-defined paths and exploit codes that target well-known vulnerabilities. Thus, honeypots that mimic a specific device do not match the device under attack and often return error responses to received requests. Upon receiving an error response from the target system, the attacker determines that it is not vulnerable and terminates the attack.

However, some attack scripts determine whether to continue the attack based on simple information contained in the response. For example, the proof-of-concept codes[2][3] for TP-Link firmware determine the device type by simply checking

[2]CVE-2017-11519, https://github.com/vakzz/tplink-CVE-2017-11519

[3]CVE-2012-5687, https://nmap.org/nsedoc/scripts/http-tplink-dir-traversal.html

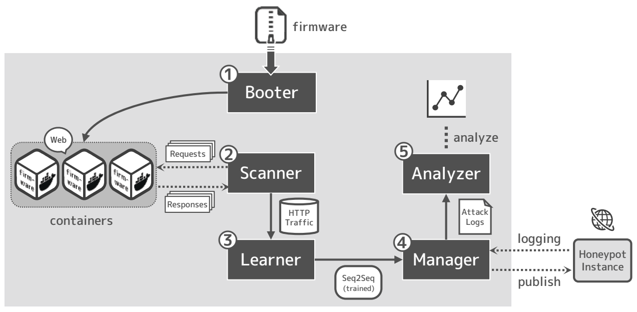

Fig. 1. A high-level overview of FirmPot.

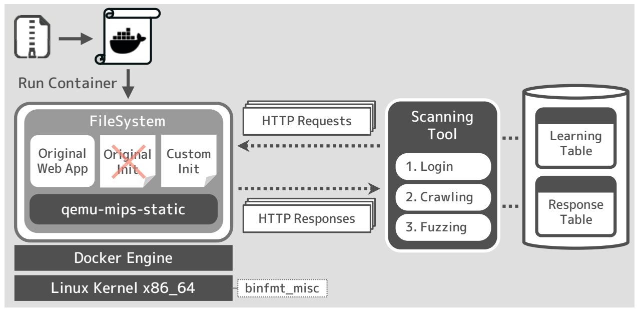

Fig. 2. The left half shows the structure of the container created by the booter, and the right half shows the collection of web communication by the scanner.

if the status code is “200” and the body contains a word such as “success”. This example suggests that attackers expect a success response and continue to attack by receiving it.

Several approaches have been proposed to analyze the received requests and respond intelligently to deal with such cases. Small et al. [14] propose to generate responses to received requests by natural language processing. However, it is unsuitable for generating vendor-dependent responses of considerable sizes, such as those issued by embedded web applications. Yagi et al. [15] defined an algorithm to convert the requested path into a honeypot’s path structure. However, their method does not take into account a query parameter and a request body. Our third challenge is to prepare a mechanism to return an optimal response continuing the attacks based on the entire information in the received request.

IV. FirmPot

This section proposes FirmPot, a framework that automatically generates intelligent-interaction honeypots that mimic embedded web applications’ behavior. The concept of FirmPot is not to work exactly like the actual system but to return the best from the pre-emulated responses to the received requests. It is based on the fact that simple information in the response triggers the attacks. For our experiment, we choose the firmware images for wireless routers.

A. High-level Overview

Fig. 1 shows the overall framework. The user prepares a firmware image and submits it to the framework, and then it automatically starts generating and operating a honeypot. The framework has five components: the *booter*, *scanner*, *learner*, *manager*, and *analyzer*. The *booter*, *scanner*, and *learner* modules are used to emulate the firmware and learn its behavior. The *manager* module is responsible for preparing an instance of honeypot and publishing it to the Internet. Moreover, the *manager* monitors the honeypot for outages and periodically collects the attack log to the local machine. The *analyzer* module is responsible for analyzing the logs received from the *manager*. For example, it outputs the statistics of received requests and the session length with clients. The results of analysis by the *analyzer* are shown in SectionV-C.

B. Booter

The *booter* provides the optimal emulator in honeypot generation for our first challenge. The goal of this module is to run the web applications embedded in firmware on a docker container. A docker container is a lightweight virtual environment in which the network and filesystem are separated from the host and are created from a docker image that specifies the configuration and startup behavior. Our framework creates a docker image that contains the filesystem extracted from the firmware image and the QEMU user-mode emulator, as shown in the left half of Fig. 2. Using binfmt_misc in the Linux kernel to launch the QEMU user-mode emulator depending on the binary format allows the container to run the executable on a different CPU architecture than the host machine.

The container to be launched receives its IP address from the docker network and executes a custom init script that the *booter* module will create for each firmware image. This custom script follows the boot process of the open-source Linux distribution provided by the OpenWrt project[4]. In the OpenWrt boot process, /sbin/init is launched first, and then the /etc/init.d/rcS script is called. /etc/init.d/rcS executes various vendor-created scripts under the /etc/rc.d/ directory, such as password initialization, TLS configuration, and web application startup. However, executing all these scripts will crash the system. The reason is that some of these scripts access network interfaces and hardware that the user-mode emulator cannot handle. Therefore, our custom scripts will not run scripts that contain words related to network or peripherals, such as “wifi”, “switch”, “led”, and so on. This technique allows us to start the web application without accessing the network settings or peripherals, thus avoiding crashes.

Since this method is based on the OpenWrt specification, it may seem that only partial firmware is supported. However, many vendors have developed using OpenWrt, and we have found that our emulator can run various firmware images. Another advantage of our emulator is the ability to launch multiple containers from a single docker image. In the next scanning phase, we reduced the time required for honeypot generation by parallelizing containers (see Section V-B).

[4]https://openwrt.org/docs/techref/initscripts

TABLE I
EXAMPLE OF A TABLE USED FOR LEARNING.

method	path	query	headers	body	res_id
POST	/login.html	a=1	Conn···	Pass···	1
GET	/favicon.ico	-	Acce···	-	2
GET	/index.php	b=2	Cont···	-	3

TABLE II
EXAMPLE OF A TABLE USED FOR HONEYPOT RESPONSE.

res_id	status	headers	body
1	200	Set-cookie···	<html >Welcome···
2	404	Content-le···	Page Not Found.
3	503	Last-modif···	503-Service Unava···

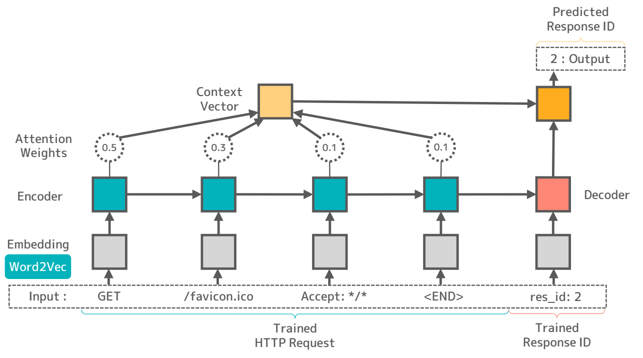

Fig. 3. Structure of our interaction model.

C. Scanner

The *scanner* solves our second challenge of mimicking the entire web application. This module is responsible for obtaining the correspondence between various requests and responses needed to mimic the behavior. The *scanner* contains a scanning tool and a database, as shown in the right half of Fig. 2. The scanning tool has three components: a login component to authenticate the user, a crawling component to explore static pages, and a fuzzing component to get dynamically changing responses. Requests sent by the scanning tool and responses returned by the web application are stored in a database.

The login component inputs the specified username and password into the forms and clicks the submit button. Cookies got upon successful login are used for crawling and fuzzing.

The crawling component searches pages in two ways. One is to search for anchor links in the retrieved web page and recursively repeat to access the retrieved ones. The other method is to use the files' path information in the filesystem's web application directory extracted from the firmware image. By combining these two methods, the crawling component retrieves as many pages as possible in the web application.

The fuzzing component collects dynamic contents by varying the request header. This component creates many requests with different combinations and values of header fields and sends them to the web application. The increase in the requests by fuzzing has resulted in an increase in training data.

There are two tables in the database for storing the collected requests and responses: the learning table, which is used during the next learning phase, and the response table, which is used when the honeypot interacts with attackers. The learning table shown in Table I contains the correspondence between requests and responses. This table contains the method, path, query, header, and body of the request, and the corresponding "response ID" (res_id). The response ID is a single number that corresponds to a single response. The correspondence between the response ID and the response is recorded in the response table shown in Table II. The date, time, and IP address in the header and body in each request or response recorded will be deleted.

D. Learner

The *learner* realizes our third challenge, a mechanism to return proper responses to attack requests. The goal of this module is to select the one from the hundreds of responses collected by the *scanner* that is likely to continue interacting with attackers. However, since an attacker's method is a black box, we do not know the correct response. Therefore, we focused on the mechanism of AI chatbots. The chatbots achieve human-level communication by learning the correspondence between questions and answers. Our idea is to create a chatbot that views HTTP protocol requests and responses as a conversation and selects a natural response to an attack request.

Our proposal uses a retrieval-based model that provides the best response from the pre-defined responses. Specifically, we create a many-to-one Sequence-to-Sequence (Seq2Seq) model with an attention mechanism that predicts the response ID from the received request recorded in the learning table.

The architecture of this model is shown in Fig. 3. Initially, the *learner* module creates a dictionary that maps words to numerical values. For the method, path, query, and body of the request, it maps each string to a single value, and for the header, it maps the key-value set of each field to a single value.

Next, a word embedding is trained from all the numerical requests by the word2vec algorithm. The trained word embedding converts similar words into similar vectors. The vectorized requests are encoded by a GRU-based Recurrent Neural Network (RNN) in Seq2Seq. Additionally, an attention mechanism assigns an attention weight to each input word, which is used to predict the output of the decoder-based RNN.

The trained Seq2Seq model is used to interact with attackers in the honeypot instance. However, real-world honeypots often receive requests that contain unlearned words, called Out-of-Vocabulary (OOV). Therefore, we prepare the algorithm proposed by Patel et al. [16] for handling OOVs. This algorithm calculates a vector of an OOV based on the strings learned by word2vec. Thus, the model treats the OOVs as words that are similar to the words already learned. With the addition of this algorithm, we achieve a honeypot that can respond intelligently to non-emulated requests.

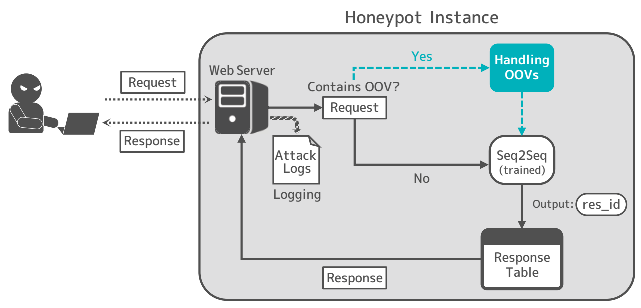

Fig. 4. Response mechanism for the honeypot instance.

TABLE III
EXPERIMENT SETUP.

Hardware		Software	
OS	Ubuntu 20.04.3 LTS	QEMU	4.2.1
Kernel	Linux 5.8.0-48 x86_64	Docker	19.3.13
CPU	i7-10700K@3.80GHz	Binwalk	2.2.1
GPU	GeForce RTX3060	SeleniumWire	2.1.2
Memory	32GB	TensorFlow	2.3.1

TABLE IV
VENDORS OF SUCCESSFULLY EMULATED FIRMWARE IMAGES.

Dataset	Vendor	Images	Success
FirmAE Dataset	Netgear	22	19
	TP-Link	18	18
	TRENDnet	12	10
	ZyXEL	6	5
Our Samples	GL.iNet	67	67
	ELECOM	21	20
	Linksys	4	4
	Buffalo	2	2
Total		152	145

TABLE V
CPU ARCHITECTURES OF SUCCESSFULLY EMULATED FIRMWARE IMAGES.

CPU Arch	bit	endian
MIPS	32	big, little
ARM	32	big, little
PowerPC	32	big
Intel 80386	32	little
MIPS64	64	big, little
Aarch64	64	little
x86-64	64	little

E. *Honeypot Instance*

A honeypot instance that our framework will eventually generate contains the web server, the trained model, and the response table. The instance is configured with an IP address and open port and published on the Internet.

Fig. 4 shows how the honeypot interacts with a client. First, the request received by the web server is checked to see if it contains OOVs. According to the result, appropriate preprocessing is applied and input to the trained model. Then, using the response ID predicted by the trained model as the primary key, it looks for the corresponding response in the response table. The web server returns the client the selected response with the IP address and hostname replaced with information of the honeypot. Finally, the received request and the predicted response are logged for analysis.

V. PERFORMANCE EVALUATION

This section shows the evaluation of our framework. The experiment setup is shown in Table III.

A. *Support Firmware*

First, we used the dataset provided by FirmAE [17] and the samples that we collected from the vendors' websites. The FirmAE dataset contains 1,124 firmware images for wireless routers and IP cameras, of which we used all 58 OpenWrt-based images. All of our samples are also OpenWrt-based.

As shown in Table IV, out of 152 firmware images from 8 vendors, our emulator successfully booted 145 firmware images. Of the seven that failed to boot, one did not contain a web application, and another two failed because QEMU did not support some of the instructions embedded in the firmware. The remaining four failed due to the certificate and key used to configure TLS for the web application. However, by manually editing the configuration file of the HTTP service to disable TLS, the emulation was succeeded.

Next, we tested our emulator using the firmware images for various CPU architectures distributed by the OpenWrt official. As a result, our emulator successfully launched the web applications embedded in firmware images having the CPU architecture shown in Table V. In total, ten different CPU architectures were supported, which is seven more than the existing emulator [5], [10], [11] introduced in Section III-A.

B. *Execution Speed*

In order to evaluate the execution speed of our emulator, we used the physical device and FirmAE [11], which employs the QEMU full-system emulator. The physical device is Archer C9 V5, TP-Link. Our emulator and FirmAE run the same firmware image as the physical device obtained from the vendor's website.

The results of honeypot generation using each of them are shown in Table VI. The booting time of the web application was the fastest with our emulator, showing a difference of approximately twice that of the physical device. In addition, there was no significant difference in the time required for one communication, from sending a request to the web application to receiving a response between our emulator and the physical device. On the other hand, the communication time of FirmAE is approximately 1.6 times longer than that of the others. Furthermore, FirmAE fell into a state where communication was impossible during the honeypot generation process. Thus, our emulator is suitable for honeypot generation because it has the same communication speed as the physical device and operates more stably than full-system emulation.

Next, we compared the honeypot generation time when using our emulator and when using a physical device, and the result is shown in Fig. 5. When there was only one container, the honeypot generation time was longer than when a physical device was used. This result is related to the fact that our

TABLE VI
COMPARISON OF THE TIME REQUIRED BY OUR EMULATOR, PHYSICAL DEVICE, AND FIRMAE EMULATOR. THE TIME IS SHOWN IN SECONDS.

	Booting Time [s]	Scanning Time [s]	Communication Time [s]
Our Emulator	44.4	1815.2	0.058
Physical Device	85.2	1655.6	0.055
FirmAE	388.7	-	0.091

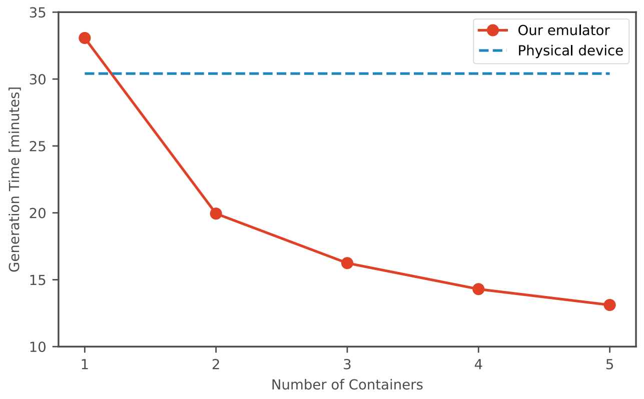

Fig. 5. Comparison of honeypot generation time between using a single physical device and 1 to 5 containers in parallel. The time is shown in minutes.

emulator has a slightly longer communication time than the physical device, as shown in Table VI. However, our approach allows us to use multiple containers in parallel to generate honeypots. As a result, the generation time for five containers in parallel was less than half for using a single container or a single physical device. Our approach is scalable because physical devices are expensive to purchase to increase the number of them.

C. Honeypot Generation and Observation

To evaluate the performance of our framework, we generated honeypots using five firmware images that were downloaded from vendors' websites. Table VII shows the firmware images used, the number of paths found by the crawling component of the *scanner*, the time to generate the honeypot, and the size of the honeypot instance. When five containers were parallelized, the shortest time to generate a honeypot was less than 5 minutes, and the longest was below 22 minutes. This time depended on the implementation of each web application. In addition, the size of instances was affected by the size of the trained model and the response table.

Next, we set up five honeypots on public servers in the university from July 1 to 30, 2021. We observed requests to the database and configuration pages in all honeypots from IP addresses different from well-known crawlers and search engines. In addition, login attempts were confirmed for honeypots No. 1 through No. 4, which have an authentication function. On the other hand, honeypot No. 5, which has no authentication function, received no login-related requests. The result suggests that attackers were aware of the difference in the interface of each honeypot.

TABLE VII
RESULTS OF THE EVALUATION WITH THE FIVE FIRMWARE SAMPLES.

No.	Firmware (Vendor)	Found Paths	Generation Time	Honeypot Size
1	Archer C9 V5 (TP-Link)	498	13 min	51 MB
2	E8450 (Linksys)	352	21 min	72 MB
3	GL-MT300N-V2 (GL.iNet)	181	9 min	17 MB
4	NBG6616 (ZyXEL)	58	4 min	37 MB
5	WRC-1167GS2 (ELECOM)	440	6 min	34 MB

Finally, we examined the session length made by each honeypot and client. If a client sends one request and the honeypot returns one response, and the communication is terminated, the session length is 1. The longer the session length, the more likely an attacker believes the honeypot is a natural system, except for DoS and brute force attacks. Hence the session length is considered one of the critical indicators of a honeypot's deception performance, i.e., the effectiveness of the learning process [8].

For comparison, we prepared rule-based low-interaction honeypots (Rulebase) in addition to our intelligent-interaction ones (Proposed). The implementation of the rule-based honeypots is based on the method of Musch et al. [13]. We also prepared a simple interaction honeypot (Simple) that responds "200 OK" to all requests.

Fig. 6 shows the session length of our honeypots and the comparison honeypots for a 30-day observation. Note that the number of clients accessing the honeypots varies by location, so the results are shown as percentages for clarity. Our honeypots had more extended interactions than other honeypots, especially the percentage of interactions that lasted longer than seven sessions.

To find out why the communication continues, we used RouterSploit[5] on our proposed honeypots and rule-based honeypots. This tool tests whether the system responds to various existing attacks. As a result, our honeypots responded to one to four attacks, while the rule-based honeypots responded to no attack. In other words, from the attacker's point of view, our honeypot appeared to be a vulnerable system that would respond to an attack, and communication could have continued. Thus, it suggests that the discussion in Section III-C that the honeypot returns the response expected by the attacker, which makes the attack continue, is correct.

VI. DISCUSSION

As a result of deploying the honeypots generated by our framework on the Internet, we observed the attacker's actions, such as login. It is not easy to observe these actions by existing low-interaction honeypots without login functions. However, we observed no advanced attack such as configuration changes. There are two possible reasons for this.

First, there is the issue of the honeypot observation location and periods. Low-profile web applications that our honeypots mimic require longer-term observation and wide-area deployment to be recognized as attack targets [13].

[5]RouterSploit, https://github.com/threat9/routersploit

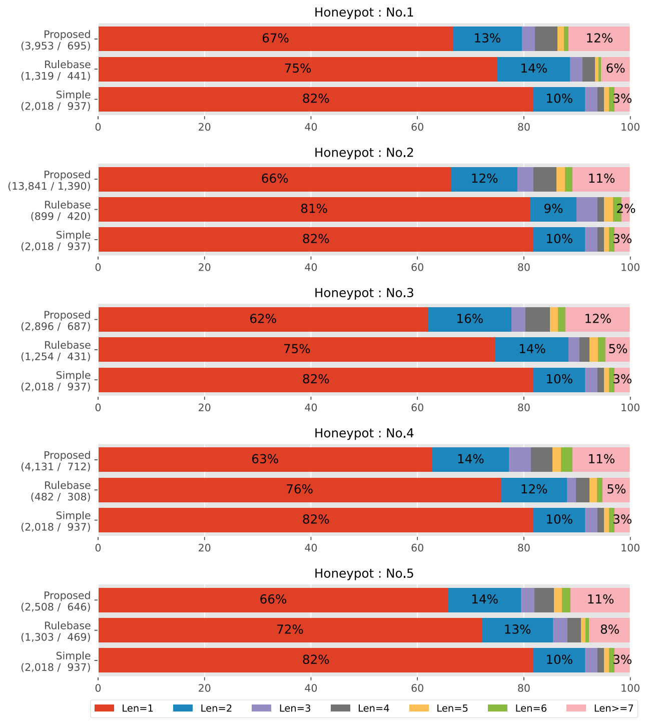

Fig. 6. Comparison in the percentage of session length with clients between honeypots. The two numbers under each honeypot method represent "(total number of requests received / total number of IP addresses observed)".

Second, our honeypots may have been detected as a decoy system by fingerprinting, which attackers did. Fingerprinting is a technique for detecting honeypots by the unnaturalness of their responses and is a common issue with all honeypots [5]. However, based on the comparison of session length showed in Section V-C, we can say that our intelligent-interaction honeypots interact with attackers more effectively than low-interaction ones. Also, our honeypots return responses based on learned knowledge without executing an attack request. Thus, unlike high-interaction honeypots, our honeypots cannot be destroyed, nor can they be hijacked by an attacker. It is safer than current high-interaction approaches for observing attacks.

VII. Ethical Considerations

The firmware images based on OpenWrt used in this experiment and evaluation are compliant with the GPL license and available from the vendor's website. The firmware image is not disclosed to the Internet in honeypot generation, and no communication with the outside world is performed. Furthermore, if the response obtained in the scanning phase contains sensitive information, it is converted to a different value to guarantee security. In the future, if a vulnerability is discovered when generating a honeypot or observing attacks, we will promptly report it to the target vendor and ask them to take action.

VIII. Conclusion

This paper presents a framework for generating honeypots using the firmware of IoT devices to observe attacks targeting embedded web applications. We have prepared a firmware emulator suitable for honeypot generation and a machine learning model that intelligently interacts with attackers. Our emulator was able to run firmware images of eight vendors and ten different CPU architectures. In addition, we generated honeypots that mimic the behavior of embedded web applications from five vendors and observed real-world attacks.

We have limited our experiments to OpenWrt-based wireless routers. In the future, we aim to realize a general-purpose honeypot generation framework that can be applied to other devices' firmware and general web applications.

References

[1] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated dynamic firmware analysis at scale: A case study on embedded web interfaces. In *Proceedings of ACM on Asia Conference on Computer and Communications Security*, pages 437–448, 2016.

[2] Wei Xie, Yikun Jiang, Yong Tang, Ning Ding, and Yuanming Gao. Vulnerability detection in iot firmware: A survey. In *International Conference on Parallel and Distributed Systems*, pages 769–772, 2017.

[3] OpenWrt. https://openwrt.org/.

[4] Juan David Guarnizo, Amit Tambe, Suman Sankar Bhunia, Martín Ochoa, Nils Ole Tippenhauer, Asaf Shabtai, and Yuval Elovici. Siphon: Towards scalable high-interaction physical honeypots. In *Proceedings of ACM Workshop on Cyber-Physical System Security*, pages 57–68, 2017.

[5] Alexander Vetterl and Richard Clayton. Honware: A virtual honeypot framework for capturing cpe and iot zero days. In *Symposium on Electronic Crime Research*, pages 1–13, 2019.

[6] Muhammad A Hakim, Hidayet Aksu, A Selcuk Uluagac, and Kemal Akkaya. U-pot: A honeypot framework for upnp-based iot devices. In *International Performance Computing and Communications Conference*, pages 1–8, 2018.

[7] Meng Wang, Javier Santillan, and Fernando Kuipers. Thingpot: an interactive internet-of-things honeypot. *arXiv:1807.04114*, 2018.

[8] Tongbo Luo, Zhaoyan Xu, Xing Jin, Yanhui Jia, and Xin Ouyang. Iotcandyjar: Towards an intelligent-interaction honeypot for iot devices. *Black Hat*, pages 1–11, 2017.

[9] Ye Zhou. Chameleon: Towards adaptive honeypot for internet of things. In *Proceedings of ACM Turing Celebration Conference - China*, pages 1–5, 2019.

[10] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS*, volume 1, pages 1–1, 2016.

[11] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In *Annual Computer Security Applications Conference*, pages 733–745, 2020.

[12] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. Firm-afl: High-throughput greybox fuzzing of iot firmware via augmented process emulation. In *USENIX Security Symposium*, pages 1099–1114, 2019.

[13] Marius Musch, Martin Härterich, and Martin Johns. Towards an automatic generation of low-interaction web application honeypots. In *Proceedings of International Conference on Availability, Reliability and Security*, pages 1–6, 2018.

[14] Sam Small, Joshua Mason, Fabian Monrose, Niels Provos, and Adam Stubblefield. To catch a predator: A natural language approach for eliciting malicious payloads. In *USENIX Security Symposium*, volume 171184, 2008.

[15] Takeshi Yagi, Naoto Tanimoto, Takeo Hariu, and Mitsutaka Itoh. Intelligent high-interaction web honeypots based on url conversion scheme. *IEICE transactions on communications*, 94(5):1339–1347, 2011.

[16] Ajay Patel, Alexander Sands, Chris Callison-Burch, and Marianna Apidianaki. Magnitude: A fast, efficient universal vector embedding utility package. *arXiv:1810.11190*, 2018.

[17] FirmAE Dataset. https://github.com/pr0v3rbs/FirmAE.